Tech Newsletter

Golden Bits Software, Inc.

Volume 1, Issue 1, Fall 2002

In This Issue:

- Resource Locking for device drivers *(with sample code)*
- Linux SNMP primer

Golden Bits is a software engineering firm providing consulting services in a wide range of diverse technologies:

- Windows, Linux, Unix
- Device Drivers, Embedded Systems
- Database, TCP/IP, GUIs

See page 8 for past projects
www.goldenbits.com

Technologies:

- ***Fibre channel***
- ***Device drivers***
- ***COM/DCOM***
- ***Database, SQL***
- ***C/C++***
- ***SCSI***
- ***GUIs/MFC***

Golden Bits Software, Inc.
3525 Del Mar Heights Rd.
Suite 158
San Diego, CA 92130
858-259-3870 ph
858-259-7655 fax
deang@goldenbits.com

Resource Locking

Interesting locking mechanisms available to writers of Windows device drivers.

Introduction

Yes, there's more to locking and synchronization than the usual mutex or spinlock approach. We all need to expand our horizons a little, so here's a look at a very cool set of APIs perfectly designed for resource locking - there's more than one way to lock a resource!

Resource locks vs. other types of locks

One of the key differences between resource locking and other locking methods, is the ability to lock a resource as exclusive or shared. Exclusive locking behaves just like your standard Mutex or Spinlock, once you have the lock no one else can get access until you release the lock. Shared locking enables more that one thread to lock the resource.

download sample code from www.goldenbits.com

Another key difference is the priority that an exclusive lock has over a shared lock, any pending shared lock requests are preempted by an exclusive request. For example, if you have a resource currently exclusively locked with two pending shared locks, and another exclusive lock request occurs, the newly arrived exclusive lock request will be given the lock BEFORE the shared lock requests. This is great if you have a common data structure that needs to be updated quickly, but you have to be careful of not starving out shared locks. You can control this priority behavior by using the functions **ExAcquireSharedStarveExclusive()** or **ExAcquireSharedWaitForExclusive().** Both of these functions enable a shared lock request to acquire a lock before any pending exclusive requests. So depending on the locking

continued on page 2

Linux SNMP Primer

An introduction into implementing SNMP on a Linux system.

SNMP, Simple Network Management Protocol, has been a standard for many years and is widely deployed on almost every imaginable device - from network routers to toasters. This article provides a jump start for implementing SNMP on a Linux based system using the NET-SNMP Open Source project. I'll cover SNMP itself (briefly), NET-SNMP open source project, implementation things, and a list of SNMP resources.

The SNMP Standard

The SNMP v1 standard was first introduced in 1990, subsequent revisions to the standard are v2 (1995) and v3 (2002). The major differences between the versions are described in **Table 2.**

continued on page 6

Resource locking (cont.)

functions and how they are used, you can starve both shared and exclusive lock requests. (Yep, we can still shoot ourselves in the foot!!)

A resource is described by the resource type, **ERESOURCE**, what's interesting is the contents of this data type (shown below):

```
typedef struct _ERESOURCE {
    LIST_ENTRY SystemResourcesList;
    POWNER_ENTRY OwnerTable;
    SHORT ActiveCount;
    USHORT Flag;
    PKSEMAPHORE SharedWaiters;
    PKEVENT ExclusiveWaiters;
    OWNER_ENTRY OwnerThreads[2];
    ULONG ContentionCount;
    USHORT NumberOfSharedWaiters;
    USHORT NumberOfExclusiveWaiters;
    union {
        PVOID Address;
        ULONG_PTR CreatorBackTraceIndex;
    };
    KSPIN_LOCK SpinLock;
} ERESOURCE, *PERESOURCE;
```

Notice the other Kernel synchronization objects: KSPIN_LOCK, KEVENT, and KSEMAPHORE. So what's happening here? Without the actual source code from Microsoft we can not be 100% certain, but it is very clear that the resource locking code builds upon the existing locking/synchronization objects. Unlike mutexes and spinlocks, resource locking functions are implemented by the kernel executive (hence the **Ex** prefix) instead of the kernel itself. This gives us some insight into how Microsoft organizes the OS internals and reinforces our notion that resource locks are built from the existing set of synchronization primitives.

Thus, there's no resource object per se. Essentially resource locking is synthesized from existing synchronization objects. Keep this in mind when you work with other operating systems that do not have resource locking functions (i.e. Linux), you can always write your own.

Resource Locking Functions

Several of the resource functions are obsolete and have been replaced by a 'Lite' version. No beer comparisons intended, but presumably these 'Lite' functions are much better (i.e. *less filling and still tastes great*); you should use them. As described earlier, a resource is represented by the **ERESOURCE**, which needs to be allocated by the calling driver. Before use, resources need to be initialized and when finished they also need to be deleted (**ExDeleteResourceLite()**). This is especially important if your driver supports dynamically loading and unloading. A very nice feature of the resource acquire functions (**ExAcquire….()**) is the ability to specify if you want your code to block by using the Wait argument. This is much cleaner than checking if you can acquire the resource and then do the actual acquire.

Two interesting functions, **ExGetSharedWaiterCount(), ExGetExclusiveWaiterCount(),** return a count of the number of threads waiting for a shared or exclusive resource. This is very useful if you're concerned about resource contention, possibly in some sort of load balancing scenario. Another function which comes in very handy in a load balancing scenario is **ExAcquireSharedStarveExclusive().**

There are a total of 18 current (not counting obsolete) resource locking functions, which can be organized as such:

Exclusive:

ExAcquireResourceExclusiveLite()
ExIsResourceAcquiredExclusiveLite();
ExConvertExclusiveToSharedLite()

Shared:

ExAcquireResourceSharedLite()
ExAcquireSharedStarveExclusive()
ExAcquireSharedWaitForExclusive()
ExIsResourceAcquiredSharedLite()

continued on page 3

Resource locking (cont.)

Other nifty functions:

ExIsResourceAcquiredLite()
ExTryToAcquireResourceExclusiveLite()
ExGetExclusiveWaiterCount()
ExGetSharedWaiterCount()

Misc:

ExInitializeResourceLite()
ExReinitializeResourceLite()
ExGetCurrentResourceThread()
ExReleaseResourceLite()
ExReleaseResourceForThreadLite()
ExSetResourceOwnerPointer()
ExDeleteResourceLite()

Obsolete functions – do not use!!

ExReleaseResource()
ExAcquireResourceExclusive()
ExAcquireResourceShared()
ExConvertExclusiveToShared()

One of the most confusing aspects of resource locking is understanding when your code will be granted the resource lock., **Table 1** should help.

Function	Mode	Description
ExAcquireResourceExclusiveLite	Exclusive	Preempts pending shared requests, waits for any exclusive or shared lock to be released.
ExAcquireResourceSharedLite	Shared	If no pending exclusive request, shared access granted immediately. If lock currently held exclusive then will pend.
ExAcquireSharedStarveExclusive	Shared	Preempts any pending exclusive requests. If lock currently held shared, access granted immediately. If lock held exclusive, then waits for release and then granted shared access before any exclusive requests.
ExAcquireSharedWaitForExclusive	Shared	Same as ExAcquireResourceSharedLite () but can optionally wait for any pending exclusive requests.

Table 1 - Summary of locking features

Sample Driver Code – An Allocator

So what's an engineer to do with all of these resource locking options? How can these functions be useful? If your system has a fixed number of resources (such as packets, tokens, bandwidth, maybe your hardware device can only handle a fixed number of requests), these functions are perfect.

To demonstrate how the resource locking functions work, a sample driver is presented (the full source code can be downloaded from: www.goldenbits.com/newsletters/issue1/resourcesample.zip). In our sample, several threads will be contending for a limited resource – a work packet. The threads will get their work packet from a set of allocator functions, it is these function that will make extensive use of the resource locking functions. A common structure, **PoolLoadBalanceInfo**, is used to load balance between all of the resource pools. Allocation requests are put on a request queue, from which each allocator thread gets a request, checks the load balancing structure for the appropriate resource pool, and then performs the actual allocation. Requests to free an allocation are not queued, they are handled immediately.

The **Figure 1** block diagram illustrates the sample driver. The key items here that illustrate the use of resource locking functions are the load balancing structure, **PoolLoadBalanceInfo,** and resource pools. The key function to examine is **AllocResource()** (see **Figure 2**). When an allocation occurs the **PoolLoadBalanceInfo** is first locked with shared access since at this point we are just reading the load balancing information. After the appropriate pool has been identified, the code does the actual allocation (gets a resource), if the allocation is successful, then the **PoolLoadBalanceInfo** is locked again, but this time in exclusive mode because we're updating the pool usage count. When a resource is freed, the structure is exclusively locked.

To keep things interesting, the sample application also gets allocation information (see **GetResourceStats()**), since this information is read only, the **PoolLoadBalanceInfo** structure is locked with shared access. As you can see, the **PoolLoadBalanceInfo** structure is central in our example. It needs to be accessed by multiple threads in both shared (read access) and exclusive (write access) mode. In the **GetResourceStats()** function we can lock the **PoolLoadBalanceInfo** using **ExAcquireSharedStarveExclusive()** or **ExAcquireResourceExclusiveLite()**.

Here's the cool part… To really see how this works, compile and install the driver and application. The application stimulates a load on the driver; it constantly allocates and frees resources from the driver. You'll notice when you run the application with **ExAcquire-SharedStarveExclusive()** selection (see **Figure 3** radio button), the number of exclusive waiters increases to 10. This is the number of allocation threads!!! This tells us that every thread has an exclusive access request and is waiting for our shared access to be release, we are starving out the exclusive access requests. Now try running the application using the **ExAcquireResourceExclusiveLite()** selection. You'll notice that the number of exclusive waiters drops down to about 1 or 2, we're no longer starving exclusive access request. ■

figures and code continued on page 4

Resource locking (cont.)

download sample code from:
www/goldenbits.com/newsletter/issue1/resourcesample.zip

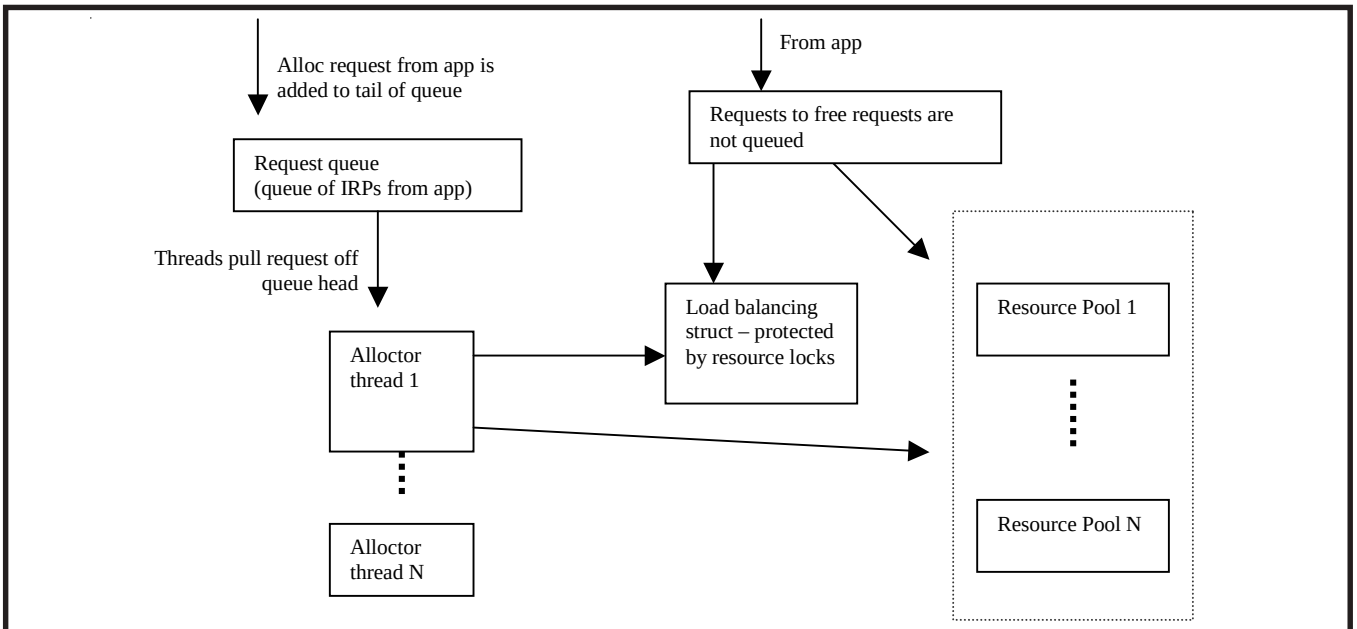

Figure 1 - Block diagram of resouce sample

```
NTSTATUS
AllocResource(
   IN  PDRIVER_ALLOCATER pDriverAlloc,
   OUT PRESOURCE_ELEMENT *pResElement
    )
{
  long lPoolID;
  PERESOURCE pLoadBalanceLock;
  PRESOURCE_POOL pResourcePool;
  NTSTATUS RetStatus;
  PLIST_ENTRY pResListItem = NULL;

    // initalize return values
    *pResElement = NULL;
    RetStatus = STATUS_SUCCESS;

    // get pointer to the resource lock for the
    // load balancing structure
    pLoadBalanceLock = &pDriverAlloc->
           PoolLoadBalanceInfo.BalanceInfoLock;

    // 1. Determine which pool to get resource
    //    from based on usesage inforation
    //    in load balancing structrure
    //    We'll ned to get a shared read lock
    //    on structure first

    ExAcquireResourceSharedLite(
         pLoadBalanceLock, TRUE);
```

Figure 2 - AllocResource()

```
    lPoolID = GetPoolToUse(&pDriverAlloc->
                PoolLoadBalanceInfo);

    // release shared lock
    ExReleaseResourceLite(pLoadBalanceLock);


    // need to turn off APCs
    KeEnterCriticalRegion();

    // get pointer to selected pool
    pResourcePool = &pDriverAlloc->
                    ResPools[lPoolID];

    // 2. After determinine which pool to get
    //    resouce from, then get exclusive
    //    resouce lock for that pool
    ExAcquireResourceExclusiveLite(
        &pResourcePool->PoolResourceLock,
          TRUE);


    // 3. After we get the excluive lock then
    //    get a resource element off the queue.
    //
    //    If resouce queue is empty then return
    //
    //    It's up to the calling thread to re-try or
    //    return an error to the application
    //
```

Figure 2 - AllocResource()

continued on page 5

Resource locking (cont.)

```
if(!IsListEmpty(&pResourcePool->ResPoolHead))
 {
    pResListItem = RemoveHeadList(
              &pResourcePool->ResPoolHead);

    // sanity check
    ASSERT(pResListItem != NULL);

    *pResElement = (PRESOURCE_ELEMENT)
                 CONTAINING_RECORD(pResListItem,
                                   RESOURCE_ELEMENT,
                                  ElementListEntry);
 }
 else
 {
    RetStatus = STATUS_INSUFFICIENT_RESOURCES;
 }

 // release lock on pool
ExReleaseResourceLite(&pResourcePool->
                   PoolResourceLock);


 // update load balancing struct only if
 // we successfuly allocated a resource
if(NT_SUCCESS(RetStatus))
 {
    // 4. Re-aquire lock on the load balancing
    //    structure, but this time we need
    //    exclusive access because
    //    we're updating the structure.
    //    This also enables us to update the
    //    load balancing info **BEFORE**
    //    other threads try to read it
    //    in shared mode.
```

Figure 2 - AllocResource()

```
        ExAcquireResourceExclusiveLite(
              pLoadBalanceLock, TRUE);

        // update the usage count, increment
        // oustanding request
        // count in the load balancing struct

        UpdatePoolUsageInfo(
             &pDriverAlloc->PoolLoadBalanceInfo,
              FALSE, lPoolID);

        // release lock on load balancing struct
       ExReleaseResourceLite(pLoadBalanceLock);
     }

    KeLeaveCriticalRegion();
    return RetStatus;
  }
```

Figure 2 - AllocResource()

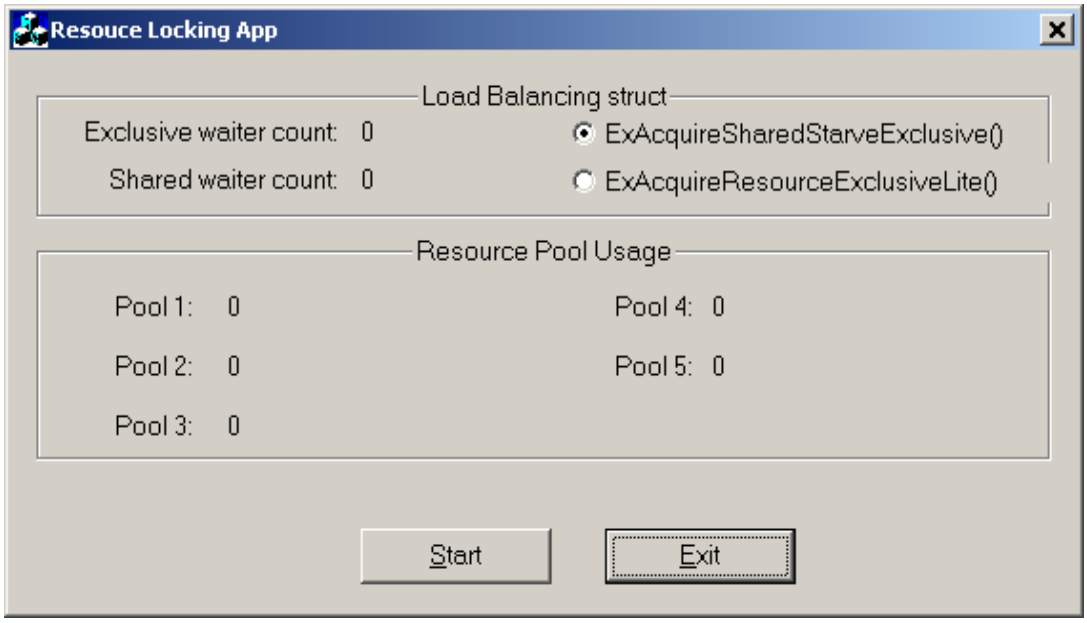

Figure 3 - App used to allocate work packets

SNMP (cont.)

SNMP v1	First standard, security based on communities.
SNMP v2	Added new OIDs for better management capabilities.
SNMP v3	Added security and encryption based on 56 bit DES. Roles and passwords used to protect data and limit access to set commands.

Table 2 - SNMP Versions

SNMP can run over a variety of network protocols, but the most common implementation is over TCP/IP using ports 161 and 162.

Important points about SNMP:

- SNMP (over TCP/IP) uses the connectionless UDP protocol, which ***does not*** guarantee packet delivery, only "best effort". Thus a trap condition may not be received by the NMS - Network Management System (HP OpenView, What's Up, or other).
- Most NMS poll the SNMP agent, generating additional network traffic
- **There is no security for SNMP v1 and v2.**

A cornerstone of SNMP is the MIB. The MIB itself is a text file that describes your management objects (things you're interested in, such as counters, levels, and status). Each management object is identified by an OID (Object Identifier), the OIDs themselves use a "dot" notation to form a tree structure. **Figure 4** illustrates the OID tree structure.

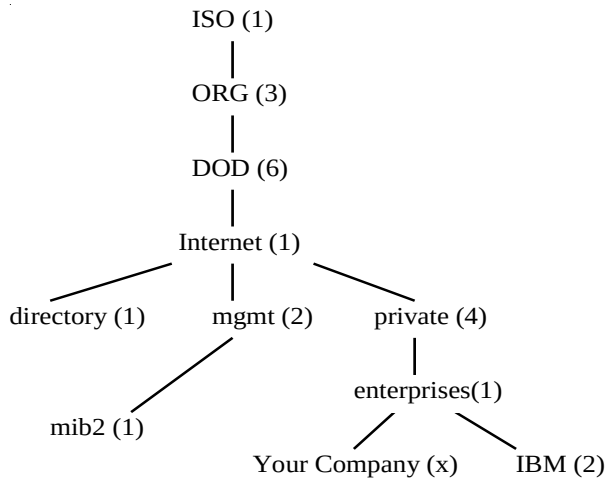

Figure 4 - OID Tree structure

All private company management data is located under the iso(1).org(3).dod(6).internet(1).private(4).enterprises(1) sub tree. For example: 1.3.6.1.4.1.2 is for IBM, 1.3.6.1.4.1.311 for Microsoft, and 1.3.6.1.4.1.71 for NASA. IANA (Internet Assigned Numbers Authority – www.ianan.com) is responsible for managing and assigning OIDs for each company or organization. You can apply on-line for your OID number at: http://www.iana.org/cgi-bin/enterprise.pl. MIB files are published by vendors to describe the management data a particular product supports. For example, Cisco publishes all of their product MIBs on their web site at: www.cisco.com/public/mibs.

NET-SNMP

Traditionally your implementation choices were build it or buy it. However, since Linux the number of solid open source projects has grown tremendously. You now have a third choice – open source. Open source offers a powerful alternative to the traditional buy vs build decision. You can leverage the work of other engineers by using an open source solution and the best part – it's free. Well almost free (yes there's no such thing as a free lunch and open source solutions are no exception), you'll probably have to do a little tweaking to get the source compiled and any bug fixes or upgrades must be shared. Generally speaking these are small things compare to the value of the shared source.

For Linux the most popular open source solution is NET-SNMP. The NET-SNMP open source project (www.net-snmp.com) was originally started at UC Davis and previously known as UCD-SNMP. While there are lots of vendors that supply SNMP solutions, the NET-SNMP project is very good; unless there's a specific need, this source base is perfect. NET-SNMP includes test tools, samples, documentation, and web references; it is essentially a full featured development kit. The current stable version is 4.2.6 (Oct 11, 2002) and it supports SNMPv1-3 protocols, and supports the following platforms: Linux, Unix, HP Unix, IBM AIX, Solaris, OpenBSD, and Windows.

Building SNMP Daemon

The first thing you should do is build the SNMP daemon; which will require your to generate the **config.h** file. This header file is used by the source tree to compile in various options and define platform specific code (i.e. Linux vs Unix), the utility, **configure**, is provided to help you with this task. The **configure** utility really does its best to create a correct **config.h**, but you will probably have to go through it by hand to verify that all the #defines are correct for Linux.

continued on page 7

SNMP(cont.)

Once you've got the **config.h** file setup correctly, then using the **make** command, from the root directory of the project, type **make**. The make files are setup to traverse all of the sub directories and build all of the necessary libraries and binaries. When finished, you should have an SNMP daemon (**agent/snmpd**) and a set of utilities (under **apps**) sub directory. At this point you should be able to startup the daemon and use the utility, **snmpget** to actually get some MIBII data from the SNMP daemon. Also make sure that any firewall settings (firewall is usually part of Linux installation) are disabled or at least enables packets through ports 160 and 161.

Hint: After you've got the **config.h** file setup the way you want, save it with a different name, such as **config.h.backup**. Otherwise when you run configure again, it will *OVERWRITE* the **config.h** file you spent so much time updating!!

Your SNMP Agent

Your management objects are implemented as an "agent" of the SNMP daemon, ***your agent is your implementation of your MIB***. This is how you add your specific management objects (things your interested in exposing – counters, status, etc.) for your new hardware or software. NET-SNMP already comes with built in agents for all of the standard MIBII objects, you get this for free; all of the agents are in the sub-directory **agent.** There are two approaches to adding an agent: 1) the agent is compiled into the main **snmpd** executable or 2) compiled as a separate loadable module that is loaded into the **snmpd** at runtime. The advantage of using the loadable module is you can update your agent by replacing only the loadable module and if there are other agents (from different vendors), you will not blow them away (this assumes the other vendor's agents are also implemented as loadable modules). I suggest you write your agent (and make file) in such as way to enable both approaches.

To create an agent, use the provided MIB compiler to generate a skeleton C file; your job is to take this file and add your specific MIB information. Don't be surprised if this C file doesn't compile right off the bat (remember, it's free software), you will have to edit the file and comment out the offending code for now to get started. To add your C file to the project you can use the **configure** utility or add to the make file by hand. Your agent file should be placed in the sub directory **agent/mibgroup**.

The layout of your agent's C file is straight forward. A table containing all of your individual MIB objects is located at the beginning of the file followed by the function calls the SNMP daemon will call to get your MIB data.

So how does my agent get called? By an exported function that is registered with the daemon process on startup or when the agent is loaded. The agent uses the macro **REGISTER_MIB()** to let the daemon know how to call your agent code. When a SNMP network request is received, the core daemon code parses the OID, looks up the correct function to call, makes the call, and sets/gets the data.

continued on page 8

MIB Misconceptions. The MIB file itself is sometimes assumed to be the only thing necessary to implement SNMP. Build a MIB, compile it, insert it into an SNMP solution and **BOOM!!** you're done. Heck the MIB compiler does all the work, doesn't it? Well not exactly. MIB compilers differ between vendors; they usually generate something, either skeleton C file or a vendor proprietary binary format, you still have to tweak the code to add your specific information. This makes sense, because the vendor has no idea what your data is or how to get it; at best they can provide you with tools (a MIB compiler) to do a lot of the grunt work. Also, not all MIB compilers are perfect, some will compile your MIB file just fine, some will bark at the same file.

SNMP (cont.)

Implementation Things

So far so good? Now what do I need to do?

First Thing:
Get an official OID from IANA (www.iana.com), this gets you a place in the OID tree under private(4).enterprises(1) sub tree.

Second Thing:
Design a MIB. This is the time when you should focus on your specific management objects, exactly what you need to mange, monitor, configure, and trap. Brainstorm with the other people on your team (or just talk to yourself if you're a team of one), some examples are:

Counters:	Errors, bad frames, bytes transferred,
Status:	Power level, CPU usage, memory usage,
Configure:	Reset counters, set thresholds,
Traps:	Loss of signal, unrecoverable errors,

You'll also want to organize your management data into different categories as they make sense.

Third Thing:
Do one OID. Implement one very simple OID from your MIB tree. Using the **snmpget** utility to get that OID data item over the network and verify it is correct. This tests the "end-to-end" connectivity, meaning you are able to generate a full SNMP request and response.

Once you can successfully get one OID, start filling out the remaining MIB tree.

Resources:

1. NET-SNMP web site: www.net-snmp.com

2. Books on SNMP:
   Essential SNMP by Douglas R. Mauro & Kevin J. Schmidt, publisher O'Reilly
   Total SNMP by Sean Hardney, publisher Prentice Hall

3. MS-SOFT. www.mg-soft.com. Sells a variety of very good development and testing tools for SNMP. My favorite is their MIB Browser Professional which runs $425, very good investment. They also just released a Linux version of their MIB Browser.

4. NMS - Network Management Systems .
   HP/Compaq. OpenView: www.openview.hp.com
   IpSwitch. WhatsUp Gold. www.whatsupgold.com

   These are two popular products which you should insure your SNMP agent works with. ■

Project Experience - Several projects Golden Bits has successfuly completed

SCSI Port driver for Fibre Channel. Designed the operating system layer for a SCSI storage driver (XP, Win2K, NT, Linux) for a fibre channel HBA (host bus adapter – PCI/SBUS card).

Embedded Network Appliance. Developed an embedded monitoring device for web sites and/or other data center systems. The device uses uC/OS real time kernel running on Motorola ColdFire processor (MCF5206e).

WDM, NDIS Device Drivers . Developed a WDM and NDIS device driver for a prototype wireless system.

Parallel Search Engine. Developed a search engine that distributes database query to other systems; the search runs in parallel on the supporting systems and the results are written (via bulk inserts) back into the database.

Satellite Set Top Box. Developed a script language and compiler used to code the television UI (guide menus, channel select).

Camera Control. Wrote highly customized Windows user interface with special graphics and custom controls. The application presents the user with camera images with graphic information overlaid (in near real time), camera configuration information, and product inspection information.

Embedded TCP/IP Protocol Stack. Wrote a NT packet driver using NDIS driver subsystem to simulate a mobile network for a military application. The embedded stack executed under NT, and the packet driver simulated network device IO.