Porting TCP/IP to the 802.4 Token Bus LAN.

Andrew McRae

Megadata Pty Ltd.
andrew@megadata.mega.oz.au

This paper describes an implementation of TCP/IP that operates on the IEEE 802.4 Token Bus LAN system. Firstly a description of 802.4 is provided, which also encompasses the motivation underlying the porting effort, then a more detailed description of the design and implementation issues is given. Finally a look is taken at the wider picture to describe the environment into which such an implementation fits.

1. The 802.4 Token Bus LAN.

The OSI standards embrace three Local Area Network media specifications, each of which cater for different environments and user needs. Each specification details the physical and electrical requirements of the LAN, as well as the method of accessing the LAN, known as the Media Access Control (MAC). The IEEE was responsible for formulating the specifications, and also for generating some ancillary standards, such as the Logical Link Control standard (802.2) that is closely associated with the LAN standards. A number of other standards are emerging that impact upon the LAN arena, such as FDDI or not-so-Local Area Networks (e.g. 802.6 Metropolitan Area Network), but these are either currently too high in connection cost or unavailable in the general computing environment to be candidates for true LAN connectivity.

The IEEE 802.3 specification (Ethernet to all intents and purposes) formalised an existing LAN defacto standard developed some time ago, which (understating somewhat) has become widespread in its installed base and vendor support. Ethernet is classed as a Carrier Sense Multiple Access/Collision Detect (CSMA/CD) technology, meaning that the LAN relies on detecting other station's transmission to indicate whether a station can transmit, whether it has to wait, or whether another station's trasmission has collided with its own. It is basically a broadcast medium, in which all stations hear all packets on the wire. Ethernet operates at 10 Megabits/second, and due to the nature of the MAC, has limits that should be adhered to for correct operation (though these limits are regularly abused).

The 802.5 Token Ring network is also designed for the office environment, and has achieved a limited success, mostly due to the support of certain large vendors. It is cabled as a physical ring, and can operate at 4 Megabits/second with later versions operating at 16 Megabits/second. The stations are connected to the ring via active transceivers. The MAC is based on a token being transmitted from each station to the next. Stations can only transmit when they 'own' the token. Some complexities arise from the need to ensure that a token must always be rotating and from token timing requirements. Token Ring has the advantage over Ethernet that it can scale to higher speeds without fundamentally changing the nature of the technology, optic fibre can be employed between stations etc, but the downside is the cabling requirements.

Both Token Ring and Ethernet are well known, and have a large installed base. The 802.4 Token Bus LAN standard is relatively unheard of in the computing community, except perhaps in the process control industry. The 802.4 Token Bus network is designed as a physically and electrically robust LAN for running the Manufacturing Automation Protocol (MAP), and is aimed at providing LAN connectivity at the 'factory floor' level. MAP is unique among protocols in that it is a standard which has been almost solely driven by a small number of specific users, the main one being General Motors. Essentially, GM generated MAP to standardise the interconnection of process control equipment, numerically controlled machining equipment and factory robots throughout a plant environment. Whilst MAP is fairly well known (if only as a buzzword), the Token Bus standard is rarely considered except as an integral part of MAP, and is relatively unknown as a LAN specification.

MAP itself is defined as a complete OSI protocol stack, incorporating 802.4 as layer 1, 802.2 LLC as layer 2, ISO CONS and CLNS as layer 3, TP4 and TP0 as layer 4, ASN.1 as layer 5, and ROSE/ACSE as layer 6. The application layers of MAP are defined as VT, FTAM and MMS (Manufacturing Messaging Service). Due to the considerable size and complexity of this requirement, and the need to fit working MAP implementations into factory equipment, a revised version of MAP (called *Mini-MAP*) was defined that removed layers 5 and 6.

Token Bus is a hybrid of LAN technologies. A range of different media, speed and cabling options are available, allowing selection of the most appropriate technology for various needs. Token Bus can operate on 10 Megabit/second broadband, 10 Megabit/second carrier band, 5 Megabit/second carrier band, and 1 Megabit/second shielded twisted pair. The physical cabling is arranged as a bus, allowing easy extension or partitioning (Figure 1). The cable used is armoured, and has extremely good electrical tolerance to noise and interference.

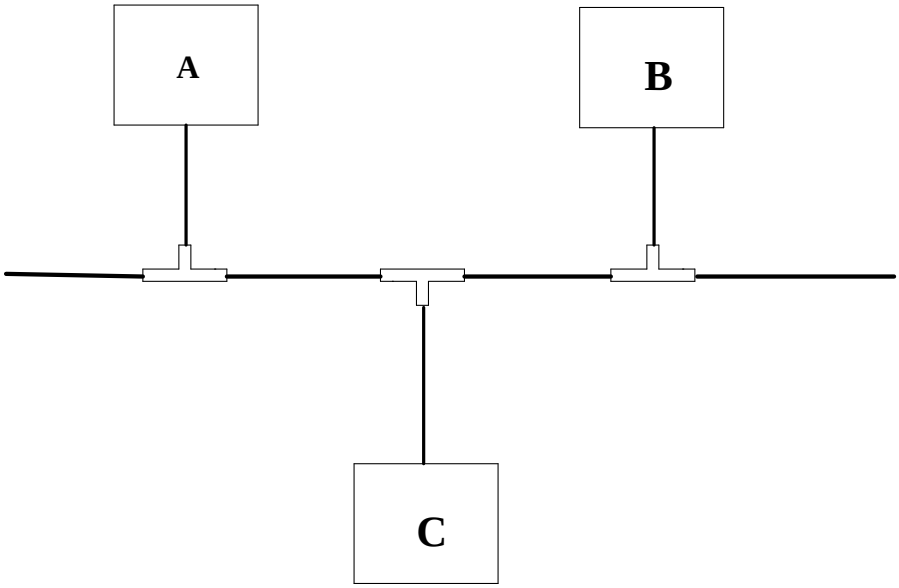

Figure 1

Each node is connected to the LAN via a passive tap. The tap is hermetically sealed and is impervious to dust and vibration. Since the tap contains no active components, much better electrical isolation can be obtained, as well as protection from LAN disintegration due to tap electrical failure.

Whilst the physical LAN is a bus, the MAC is achieved through a logical station ring, where a token is passed from station to station (Figure 2). Since the LAN is a bus, the media is a broadcast medium, but with a layer of sophisticated control dealing with the access of the stations onto the LAN via a logical token ring. This allows greater control over the operating characteristics of the LAN, both dynamic and static, as well as exceptional robustness and fast response to error conditions.

Token Bus demands quite a bit more from a controller than Ethernet. For example, the Token Bus controller after passing a token to the next logical station will eavesdrop on the LAN to ensure that the token pass was successful. If not, it retries and eventually recovers by cutting out the failed station from the ring and establishing a new ‘next station’ by passing the token to the next logical station after the failed station. Periodically a delay is inserted in the token pass to solicit for any new stations that wish to enter the ring.

The ring is established upon initialisation by stations attempting to ‘claim’ the token. The 48 bit MAC address is instrumental in providing prioritisation of the stations both with token claim and also establishing the order of the token pass (from highest MAC address down to lowest MAC address). Major

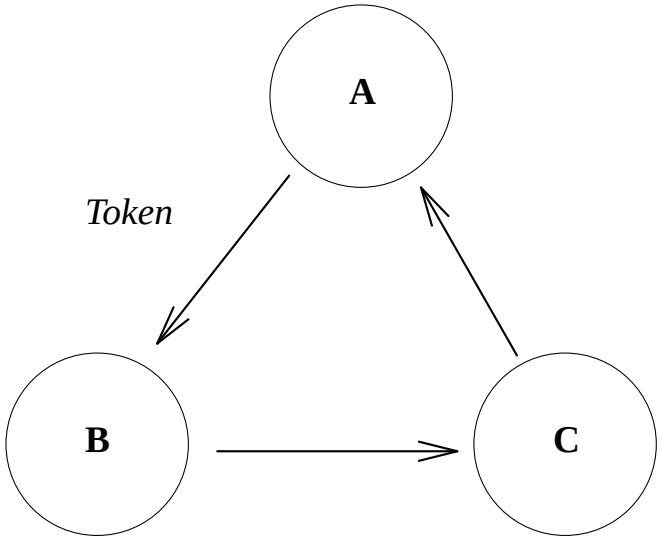

Figure 2

errors such as a station detecting another station using the same MAC address will force the offending stations into an OFFLINE state so they are not part of the ring. Stations can only transmit when they hold the token. If any station detects the collapse of the logical ring (by not hearing any token passing), that station will attempt to reinitialise the ring. The ring set up time is short enough (depending on the number of stations it can be as low as a small number of milliseconds) so that LAN access is not greatly affected in the event of a collapse and reinitialise.

Apart from the robust nature of the physical bus and the logical ring, the Token Bus LAN allows *deterministic* control over the LAN access for each station so that certain and definable minimum times can be guaranteed. A lot is said about determinism (in a computing sense, as well as theologically and philosophically), but anyone who has experienced a loaded Ethernet will know that whatever determinism is, Ethernet doesn't have it. It is widely accepted and widely misunderstood that Ethernet starts to come apart at the seams at a relatively low load, but as accurate studies have shown, Ethernet stands up well until you hit about 70%, and *then* falls apart. Token Bus as a deterministic LAN guarantees that even in the event of every station wishing to transmit lots of data, no one station will hog the LAN, and every station will be given a chance to send; moreover, the determinism is under the control of the system designer.

This attribute is a key feature of 802.4, and is a major factor in the choice of 802.4 as the preferred LAN for real time or mission critical distributed systems. How is this LAN determinism achieved?

The LAN is configured for a specified Target Rotation Time (TRT) in octets; this is the desired maximum time it takes for a token to complete a full cycle of the logical ring, expressed as the number of octets transmitted on the LAN, including all preamble and frame control octets. When a station receives the token, it considers the number of octets that have been transmitted since it received the token *last* time. This value (termed the Last Token Rotation Time - LTRT) is compared against the TRT and determines whether this station can transmit any packets. If a station has no data to transmit, it simply passes the token, so the minimum token rotation time may be much less than the TRT maximum. However if a station has transmit packets queued it will only begin the transmission of each if the number of octets that has been transmitted on the LAN is less than the desired TRT. In other words, every station monitors the traffic volume, and automatically restricts its LAN usage in the event of congestion. The net effect is an averaging of available bandwidth to all stations fairly.

As an example, say we have 5 stations, called A, B, C, D and E. The token is passed from A→B→C→D→E. The LAN has a maximum TRT of 2000 octets (not including framing and preamble octets), and each station is transmitting the following packets every rotation of the token:

Station	Packet size	count
A	500	1
B	300	2
C	100	2
D	200	1
E	-	none

The table below details each station's last measured rotation time, and the number of data octets it transmits on each rotation.

		A	B	C	D	E
1	LTRT	-	-	-	-	-
	XMIT	500	600	200	200	0
2	LTRT	1500	1500	1500	1500	1500
	XMIT	500	600	200	200	0
3	LTRT	1500	1500	1500	1500	1500
	XMIT	500	600	200	200	700
4	LTRT	2200	1700	1400	1600	1400
	XMIT	0	300	200	200	700
5	LTRT	1400	1900	1900	1800	2000
	XMIT	500	300	100	200	0
6	LTRT	1100	1100	1400	1500	1500
	XMIT	500	600	200	200	700

The first two rotations are less than the maximum TRT, so all stations transmit the data they have queued without delay. On the third rotation station E has a 700 byte packet queued for transmission; because the measured rotation time is less than the TRT, it sends the packet. However station A now has a measured rotation time of 2200 octets, and so passes the token without sending any data. Station B has a measured time of 1700 octets, and so can only transmit one of its two 300 byte packets. Station A gets to transmit on the next rotation though, whilst station E will miss out, and so on. In effect the transmitting of the data will average so that the TRT is maintained, but no station is locked out of sending data for more than one rotation

An even more bewildering factor is added when the Token Bus priority structure is employed. The Token Bus MAC defines 4 priority levels for all packets, which are termed priority 0, 2, 4 and 6 (priority 0 is lowest, 6 is most favoured). The numbering scheme is derived from the OSI 8 level priority scheme, with each 802.4 level having two OSI priority levels mapped to it. Each priority level is assigned its own TRT. Packets on priority level 6 are *always* transmitted, regardless of the measured rotation time; however a separate octet counter is kept that is a limit on the amount of time that a station can hold the token for, so it is impossible for a single station to lock out all other stations. Packets on the lower priority queues are only transmitted if the TRT for that priority level has not been exceeded. Each station may have packets on all queues, and stations may be set up with different TRT's for each priority, so it all gets rather complicated.

Is all this priority stuff useful? What happened to the *KISS* principle? When seen in the light of 10 Megabit/second Ethernet, and how much data is required to really saturate it, one could be forgiven in thinking that Token Bus is overly complex, and that if you are pumping that much data into a network, something is wrong somewhere. The short answer is that Token Bus is just what the doctor ordered for real time networks, because of the deterministic nature of the LAN. Whilst it is unlikely that current generations of products can push enough data into a 10 Megabit/second broadband Token Bus to really require tight control over the prioritisation of data, Token Bus is becoming increasingly common over slower speed networks such as 1 Megabit twisted pair, and even down to 100 Kilobit twisted pair. At these speeds, the

priority mechanism is absolutely vital so that maximum control is gained over the limited bandwidth resource. It guarantees that more important data are sent ahead of less important data, especially at times when there is a burst of LAN activity (which is often the time when the important data has to get through fastest).

2. The Target Platform.

Megadata had been working with several customers in the area of embedded distributed control in hostile environments, where intelligent nodes are dedicated to monitoring and controlling specific items of plant equipment, and these nodes are networked to provide information and control throughout the plant. It was vital that these nodes could also operate in the event of total LAN failure, so that automatic control of the equipment could still take place. One major market is in the electrical substation control area, where each intelligent node could be attached to a circuit breaker, or a transformer etc, and each node would contain the specific control algorithms necessary to operate the equipment. A major aim was to go beyond the simplistic dedicated Programmable Logic Controller (PLC) solution into a completely integrated enterprise wide control environment, where the intelligent nodes were accessible from different types of computing equipment throughout the entire network. A key element of this strategy was the involvement of Open Systems, so that customers could integrate new systems from different vendors, and be assured that it could interoperate with the control system. To do this required the implementation of standard network protocols at the lowest possible level. The question was, which protocol should be chosen?

Whilst MAP as an Open System as yet to take the world by storm, the 802.4 physical network is very attractive in hostile environments. Some time ago Megadata developed working implementations of Token Bus hardware, with the intention of eventually supporting full MAP, or even Mini-MAP. It was soon realised that a working MAP implementation would require much greater processing power, memory and other resources than was desirable in an embedded environment. A number of MAP systems already were in existence, but were usually implemented using a fast, powerful microprocessor such as 68020, and also required memory in the order of 2 to 3 Megabytes of RAM. Other vendors experienced extremely poor performance (some as low as 2 to 3 Kilobytes of data throughput *per second*) due to untuned implementations, slow CPU performance or the heavyweight nature of the protocol overhead. Another major reason not to use MAP was the problem interconnecting MAP based nodes with more common computing equipment (such as UNIX† workstations); application level gateways, transport level bridges, dual protocol stacks etc. all start to cost a lot of time and money.

The platform that Megadata developed for the 802.4 Token Bus was based on an embedded Motorola 68000 processor system, designed for low cost and low power consumption, which was called a Distributed Interface Unit (*DIU*). Two basic configurations exist; a larger card for a mid-range series of (now obsolescent) modules, and a smaller version for fitting into a physically compact, low cost range of cards. The cards were about the size of a large postcard. Each system contained a processor card, one 802.4 controller card and up to 5 data acquisition cards, all custom designed and built by Megadata.

The processor card contained a maximum of 128 Kb EPROM, 64 Kb RAM and 64 Kb non-volatile battery backed RAM. The processor was a 68000 running at 7.6 Mhz, which when combined with some EPROM, didn't exactly astound anyone with speed. I'm not even sure if it would be fast enough to even appear on a SPECmark chart (except perhaps at 300 DPI). A rough calculation indicated the processor ran somewhere about .6 MIPS (if the wind was right).

The 802.4 interface controller card was based on the Motorola 68824 Token Bus Controller (TBC). The physical LAN interface used a Siemens Carrier Band modem chip. The controller card contained 64 Kb RAM which the main processor and the TBC shared access.

The program space of 128 Kb presented a problem, as this was the total space allowed to hold not only the multi-tasking operating system and the drivers, but the networking code, the application library routines and the complete application code. The RAM available was only slightly less restrictive: the processor had access to 128 Kb onboard RAM (64 Kb was non-volatile) and 64 Kb offboard. The offboard RAM was much slower due to bus access timings, and also was shared with the TBC.

---

† UNIX is a trademark of Bell Laboratories.

It was obvious with the limited memory available that a mini-MAP system (let alone full MAP) was out of the question. Besides, it would not meet one major goal of easy interconnection with more common UNIX based computers.

The natural choice (and a much more attractive proposition) was to use TCP/IP on the DIU to both interconnect DIUs via the Token Bus, and to provide node to node connection to other computing equipment via routers and gateways. And so it was decided to port TCP/IP to the DIU embedded processor system. Another goal was to provide a reasonable Application Programming Interface via a UNIX -like real time embedded operating system, along with a complete Berkeley style networking application interface (BSD sockets).

Needless to say, this presented a challenge.

Nevertheless, the advantages gained were significant. By using the Berkeley UNIX TCP/IP software, large amounts of time could be saved by not developing and (more importantly) testing/tuning a custom TCP/IP implementation. In fact, one goal was to change as little as possible of the BSD code, so that upgrades or new code could be easily integrated with the existing code. This also opened up the future possibility of porting the soon to be released BSD 4.4 networking software, which implements the OSI protocols; thus providing a base for developing a dual MAP/IP protocol stack as MAP implementations become more widespread.

Another major advantage is to be able to utilise application code developed under UNIX directly by maintaining the same socket API. This would enable Megadata to leverage its coding efforts across a complete range of products, from UNIX workstations down to embedded systems.

Once it was decided that TCP/IP was the networking standard, it was clear that the 802.4 LAN could easily be internetworked with standard UNIX workstations via 802.4 to Ethernet gateways, or via SLIP lines, providing a completely homogenous networking environment across the entire product range. The possibilities were endless.

3. Porting the Software.

A lot of the effort was involved with developing the multi-tasking Unix-like operating system. Over the last several years Megadata had evolved a basic embedded kernel which fulfilled most of the requirements, such as multiple processes, structured device driver interfaces, memory heap management etc. It looked a lot like UNIX from the programmer's point of view, only it didn't have a file system. Since this OS was not operating on a system with memory management, the process model resembles a single Unix process running multiple threads. The threads are not pre-emptible, and share the same memory image of program and data space. The model falls somewhere in between heavyweight UNIX processes, which have no direct access to other process's memory image, and true lightweight threads, which are more intimately connected to each other (e.g. common I/O and signal handling, inter-thread control). They are called threads for want of a distinguishing term.

Threads can spawn other threads at any time, with the only limitation being available RAM. Each thread has its own stack and set of file descriptors, and access to the OS services may cause the thread to be suspended without affecting the other threads. No real signal system is supported, so this simplifies the handling of threads, also making context switching fast. The OS services such as device drivers, socket calls etc. are entered as subroutines rather than as system traps (as in a real kernel). Each thread's context is maintained in a process control block, which also contains the stack for the thread. The size of this kernel was less than 10 Kb.

An earlier project required TCP/IP support on Ethernet for an embedded product, so using the existing kernel as a base, support was added for the socket interface routines and the other support routines needed for the BSD TCP/IP code. An aim at all times was to minimise the amount of code used, but not to rewrite large amounts of code. The result is a 'lean, mean and hungry kernel', that as a networking OS complete with drivers, TCP/IP code, and socket interface code, fits under 64Kb of program space and requires less than 12 Kb RAM (apart from mbuf space). The code breakdown looks something like:

Code Use	Size (bytes)	% of Total
Vectors/debug-monitor	6730	10.4
Socket/buffering support	12258	19.0
Net I/F support	3628	5.6
Routing	1444	2.2
ARP	2792	4.3
Internet Protocol	13238	20.5
UDP	2026	3.1
TCP	11214	17.4
Operating System	8570	13.3
TBC Driver	2612	4.0

The total program space required was 64512; thus we had at least 64 Kb program space reserved for application programs, a reasonable amount considering the OS and network services available.

The first generation of the network code used an early release of the *netinet* 4.3 BSD code, without any of the *net* or *sys* code, so a lot of early work went into developing some simple routines to enable the basic TCP/IP code to operate. When the complete 4.3 BSD release became available a new version of network code was developed that incorporated the full set of BSD networking code, such as the *mbuf* routines, the routing and network interface code.

Some work went into developing an OS interface matching the BSD *socket* API; standard calls such as *socket*, *bind*, *connect*, *accept* etc. are all supported. These routines provide an interface between the application program and the BSD code actually implementing the calls. The standard BSD code for these system calls was unusable due to the fact that they are designed to operate on a typical UNIX system call interface, with a large amount of argument address checking needed to transfer data across the user/kernel mode boundary. Another interesting side effect of having a shared memory image is that when a system call such as *ioctl* is called, normally a library stub routine would be called that would perform a system trap; the trap handler would then indirectly call the handler for that service after mapping the arguments into the per process *u* area. After calling the kernel subroutine the trap handler would perform some signal processing, check to see if an error had occurred, and finally return from the trap handler to the user code. Rather than replicating this, the embedded kernel *ioctl* subroutine would be called directly; the kernel subroutine would look like this:

```
ioctl(fd, cmd, data)
int     fd, cmd;
caddr_t data;
{
struct file *fp;
int err;

        curtask->p_error = curtask->p_return = 0;
        if (fp = getf(fd))
                {
                if (err = (*fp->f_ops->fo_ioctl)(fp, cmd, data))
                        RETERROR(err);
                }
        RETURN(curtask);
}
```

To handle errors in a standard way, two macros are defined: `RETERROR`, which sets the *errno* global to the error number and returns -1, and `RETURN`, which checks the process control block (pointed to by *curtask*) error value *p_error*, and if zero returns using *curtask->p_return* as the return value, otherwise performing a `RETERR(curtask->p_error)`. The kernel is then simply a set of subroutines that the

application code calls, simplifying the code and improving the performance considerably.

Some reverse engineering had to be done for those system subroutines that the TCP/IP code called, and to provide some equivalent stub; generally if the call was not meaningful in the embedded environment (e.g. *suser()*) then the call was defined as a dummy macro. When this work was started not a great deal was known about the internels of 4.3 BSD Unix, and with some judicious poking around in kernels and guesswork most of the calls were implemented. When *The Design and Implementation of the 4.3 BSD Unix Operating System* was finally available, it was absorbed with much interest to see how much we got right (and to see which bits were luck).

Once the socket interface routines were written and tested, it was fairly easy to port the TCP/IP code directly. For such a large (> 8000 lines) amount of code, it was suprisingly portable, which is certainly a tribute to the designers. The only changes that were made was to modify some static data initialisation to enable the code to be placed in EPROM, and to reduce some table sizes to shrink the RAM usage.

The major difference between the embedded target environment and the UNIX kernel environment is the lack of hardware memory management. This touched a number of areas in the porting process. One such area (as mentioned) was the interface between what is typically the user mode and the kernel (protected) mode. In the usual kernel, the data is transferred via special subroutines that map the user data into kernel accessible data, which also check for page violation. One such routine is *uiomove*, which in the embedded system effectively maps to a simple *bcopy*. Such a simplification leads to a big win in the area of performance.

One special area of note is the *mbuf* memory buffering scheme that virtually all the socket and TCP/IP code is built around. *Mbufs* are the coin of the realm for nearly all pieces of data used in the networking code, and because of this the issue of memory management policy and implementation is neatly isolated into a small section of code.

On a normal BSD UNIX kernel, *mbufs* are allocated from a growing pool of memory pages. When the current pool is exhausted, new pages are allocated using the *mbuf* subroutine *m_clalloc*, which attempts to allocate new page table entries and obtain extra pages of memory from the system virtual memory allocation subroutines. If the operation fails or the limit of pages is reached, an attempt is made to reclaim memory by asking the protocol handlers to give up non-essential memory. A separate *mbuf* page mapping is maintained at a fixed address (like the *u* memory area).

At first glance, it looks difficult to fit this into a flat addressed non-memory managed system, but at closer inspection the *mbuf* memory can be preallocated using a fixed size at a fixed address. Most of the virtual memory calls can be eliminated, with the exception of just one (*rmalloc*) which returns a incrementing page index into the *mbuf* store. This constrained the mbuf memory to be pre-allocated and of a fixed length, but all up this wasn't seen to be disadvantageous. In fact the *mbuf* scheme has proved to be fast, efficient and flexible for the processing of network protocol data, so it was very desirable not to re-engineer the memory management (this would have led to perhaps drastic changes throughout the networking code).

4. Driving the Token Bus.

The Motorola 68824 Token Bus Controller (TBC) was the main component of the controller card. The TBC is similar to many LAN controllers of the same ilk, being fairly intelligent about chained buffers, along with the usual share of incorrect documentation, missing documentation, incomprehensible registers and needing some 'it works this way, but not that way' trial and error programming. The 802.4 specification is very complex compared to Ethernet, and there exists a large number of magic incantations that you must speak to the chip to get it to do the right thing.

Newer network interface chips are usually smart when it comes to buffer handling, and the TBC is no exception. The TBC operates via a pool of free memory descriptors, one type which is assigned for each complete data packet (containing frame type, MAC addresses etc.), and another type which is a chained list of memory buffer headers, which point to separate memory buffers. The descriptors are set up so that offsets can be used into buffers, allowing LLC headers to be easily added or removed from data frames. The controller card containing the TBC also had 64 Kb RAM available, which contained the *mbuf* store; thus the TBC had direct access to the networking data, minimising the amount of data shuffling. This was a big win on the performance side, especially when large packets are being sent or received.

One point to note is that the Maximum Transmission Unit (MTU) of 802.4 is 8 Kb, which means that a large amount of data can be pumped around the net without a lot of protocol overhead; unfortunately this can also mean a lot of IP fragmentation when gateweying via SLIP.

5. Some Results.

The first aim of the project was to functionally test the networking code using a Serial Line/Internet Protocol (SL/IP) connection. This verified the correct operation of the OS and networking code, and also allowed some measurement of performance on the embedded processor. Next, the TBC driver was developed. Luckily quite a lot of experimentation had already been performed using the TBC on a Token Bus network, so a driver was running in a matter of days.

The experimental network was set up as shown in figure 3.

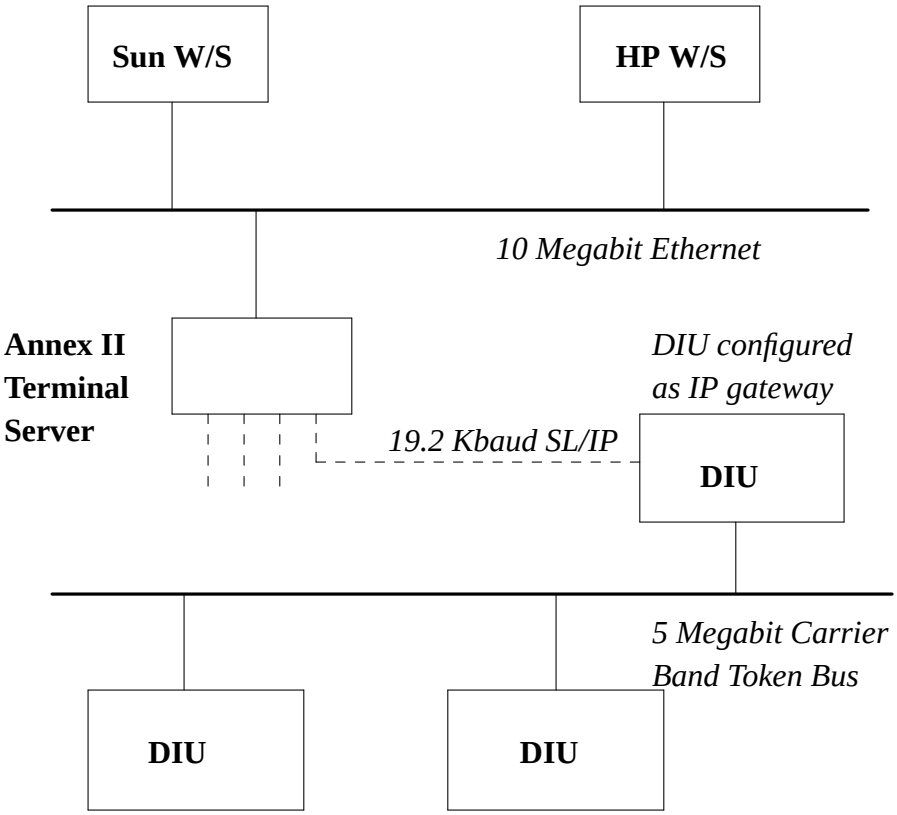

Figure 3

The use of TCP/IP to interconnect the DIU via SL/IP to a terminal server gateway onto the Ethernet immediately meant that all the network debugging tools such as *ping*, *ttcp* and others could be used from any workstation on the network. One of the first applications to be developed was a remote login facility via *telnet*, which was an excellent proof of concept test. Users could remotely log onto the DIU, and perform system configuration, examine statistics and monitor operation, all from the comfort and safety of their workstation. An example session is shown below:

```
noah% telnet map102
Trying 192.73.12.1 ...
Connected to map102.
Escape character is '^]'.
Megadata DIU Site configuration details
```

```
--------------------------------------
Software date    : Tue Apr  2 13:01:44 EST 1991
gateway          : 192.55.99.31
station-address  : 192.73.12.1
Free memory      : 64642
MD1: help
SIte ......... Site configuration
SCanlist ..... Module scanlist
MONitor ...... Enter Debug monitor
MAp .......... TBC/LAN monitoring
Connect ad ... Connect to another DIU
ARp .......... Display ARP tables
If ........... Display interface summary
NEt .......... Display network connections
MBuf ......... Display mbuf stats
Ping ......... Ping host
LCR .......... Display LCR status
DIU: if
Name  Mtu   Net/Dest    Address      Ipkts   Ierrs Opkts   Oerrs Collis Queue
lo0  1536  127.0.0.0   127.0.0.1        0       0      0      0      0      0
tbc0 8173  192.73.12.0 192.73.12.1  77874       0  15681      0      0      0
sl0  1006 192.55.99.31 192.73.12.1 856228       0 856425      0      0      0
DIU:
DIU: ^]
telnet> quit
Connection closed.
noah%
```

Further measurements indicated that a basic Remote Procedure Call (RPC) between two DIU's took between 5 and 10 milliseconds, which was pleasantly surprising given the capabilities of the CPU. A operational system that implements a complete substation control environment has been developed and is due to be installed at sites throughout Western Sydney, and in tests has shown to be capable of high speed control, data acquisition and monitoring, and yet has sophisticated facilities for connection to a Wide Area Network via TCP/IP, remote login to any DIU from any other node, and can be connected to any equipment that supports TCP/IP. The performance is entirely due to the distributed nature of the system, and to the fact that TCP/IP is a fast and lightweight protocol stack.

In such a system it is possible to build an enterprise wide network based on Open Systems protocols which interconnects the largest mainframes with the smallest embedded nodes; and if the network were attached to the Internet, you may be blaming power outages on worms instead of storms.

6. Future Applications.

A number of improvements are being investigated at the present time.

It is envisaged that the IP priority option values can be directly mapped to the 802.4 priority scheme, so that the application has direct control over the priority of the data being sent.

To further enhance the facilities available, the Sun Microsystems's RPC/XDR remote procedure call system will be ported, providing applications with a sophisticated and powerful RPC mechanism. Also in the pipeline is an SNMP agent using the standard MIB for 802.4 Token Bus interfaces, so that each node can be managed using standard network management tools.

A version of the DIU is developed that will operate on twisted pair at 1 Megabit; this will reduce the connection cost for each node, and also reduce the cabling cost. An Ethernet controller card is operational, which allows a much faster gateway between 802.4 and Ethernet to operate. The logical outcome is that Token Bus networks can be internetted to other networks, and IP packets routed across the gateway rather than slow application bridges performing data conversions. Thus it would be possible to closely link the

UNIX workstation world with the factory floor environment, facilitating a wide range of technology and computing power to control applications, was well as delivering data to MIS databases. Once MAP becomes more widespread, transport layer bridges could be used to provide a migration path for older systems.

Some investigation is under way to provide a smaller, cheaper node based on current generation embedded processors, such as the Intel i960SA/SB, which provides powerful CPU performance and floating point in a single chip designed for the embedded environment.

The ultimate goal of employing Open Systems standards is to allow users to preserve their investment (hardware and applications) by easily and simply interconnecting computing equipment and software from a range of vendors. Whilst this is starting to become a reality in the workstation and general computing fields, it is unheard of in the the real time field. What should be happening is for vendors of control and plant equipment *building in* Open Systems connectivity into products e.g. when a manufacturer sell a circuit breaker, it comes with a standard LAN connection, allowing the installer to simply plug the equipment onto the control LAN and immediately integrate it into the control network.

7. Conclusion.

The use of TCP/IP as an Open System protocol suite on the IEEE 802.4 Token Bus LAN in the real time embedded control area has proved to be a success, displacing proprietary protocols. The use of a standard LAN means that, in time to come, the physical and logical interconnection of different vendor's equipment will be easier. The porting of TCP/IP to 802.4 provides a fast and efficient networking standard which can operate on a limited hardware platform, whilst still maintaining the numerous advantages of a common networking foundation across a complete range of computing environments.