INTERFACING CAMAC TO A UNIX SYSTEM

G. LEE and D. WIEGANDT
Organisation Européenne pour la Recherche Nucléaire, CH-1211 Geneva 23 (Switzerland)
(Received July 7, 1983)

Summary

In this paper the interfacing of a CAMAC system to a PDP-11/45 running UNIX version 7 is described. The main emphasis of the paper is to describe the CAMAC device driver for UNIX and to illustrate the advantages of programming the driver in the high level language C as compared with programming in assembler. It is assumed that the readership is primarily from a hardware background, and therefore we summarize some of the background and history of the UNIX system.

1. The UNIX system and philosophy

1.1. Evolution of UNIX

The first version of the UNIX operating system was developed by Thompson of Bell Laboratories for a PDP-7 computer. Later, in conjunction with Ritchie, a second version was developed for the PDP-11/20 computer [1 - 3]. During the 1970s, UNIX for the PDP-11 range evolved through several versions with version 7 (V7) being the most widely used today and indeed the version that runs on the PDP-11/45 at the Organisation Européenne pour la Recherche Nucléaire (CERN). The latest version is system III which was released at the end of 1981 and the next version, system V, is due for release in mid-1983.

In addition to the developments for the PDP-11 range, UNIX was implemented on many other machines, notably the Digital Equipment Corporation (DEC) VAX range. 32V and system III are UNIX systems for VAX hardware and are distributed by Bell Laboratories. In a separate development at the Computer Science Division of the University of California at Berkeley, a virtual memory version of UNIX for the VAX was produced. Today, Berkeley UNIX 4.1 BSD is the most widely used UNIX system on VAX hardware.

UNIX is now available for a large range of machines from microcomputers to mainframe computers and the use and popularity of UNIX have grown enormously. Furthermore, with the advent of cheap 16-bit microcomputers and work stations, UNIX is rapidly becoming the *de facto* standard operating system for such machines.

0252-7308/83/$3.00

1.2. Some UNIX terminology

The kernel of the UNIX system is almost entirely written in the language C, a language developed from BCPL by Kernighan and Ritchie [4]. The kernel itself is relatively small in terms of code and, with simplicity an important design goal, it provides a limited well-defined set of system calls for process and file management. The term UNIX system is often used to describe not only the kernel but also the large number of utilities and application programs which make up the UNIX "tool kit" of software modules. The use of this tool kit is an important aspect of software development under UNIX.

Rather than the development of a large number of specialized software packages, the philosophy in UNIX is to take the basic software tool kit and to combine tools in order to perform more elaborate tasks. The UNIX "shell" and "pipe" mechanisms are instrumental in this combination, and it is through this modular approach to software development that UNIX achieves a large degree of flexibility.

1.3. The UNIX shell

The UNIX shell is a process which handles the user interface to the system. In addition to being the command language for issuing and executing commands, it provides a variety of control flow primitives for iteration and branching. As UNIX is an interactive system, the shell normally accepts its input from a terminal, but it can easily be redirected to take its input from a file. Files containing lists of shell commands are called shell scripts and, using the control structures mentioned above, complex programs can be developed. The shell language, although sometimes a little cryptic, is very powerful and is a flexible way of combining the software tools referred to in Section 1.2.

1.4. UNIX pipes

The UNIX pipe mechanism allows the output of one process to be used as input to another in a manner that is totally transparent to the user. Pipes make use of both shared memory and temporary files depending on the quantity of data being exchanged and all problems of data flow control in the pipe are handled by the system. Pipes provide the mechanism whereby users can easily chain together simple processes or "filters" to perform a more complex operation. It should be noted, however, that pipes are not a general-purpose interprocess communication facility since they only operate between "related" processes, *i.e.* processes which share a common ancestor.

1.5. UNIX and real-time applications

UNIX was developed primarily as a time-sharing system and is not aimed at real-time applications. Although several attempts have been made to build a "real-time UNIX" [5, 6], there are a number of deficiencies in standard UNIX which restrict its applicability in the area of real-time problems.

(1) No user-accessible mechanisms exist in standard UNIX to lock a process in memory so as to guarantee response within a given time limit to external stimuli. In keeping with its time-sharing philosophy, processes compete in an equal manner for memory resources and are always candidates for swapping.

(2) Asynchronous input–output (I/O) requests have only recently been introduced in UNIX. In V7, once an I/O request has been issued, the issuing process will wait until the request has been completed. A common requirement in real-time applications is to be able to wait on several I/O requests or events and, with the above limitation, such a requirement cannot be met.

(3) The event–wait mechanism in UNIX V7 does not associate memory (an event control block) with an event. Under these circumstances, the process waiting for an event that has already happened will wait forever as there is no way to inform the process that the event has already occurred. The semaphores and shared memory features that will be available in system V will remedy this situation.

(4) UNIX V7 lacks a general-purpose interprocess communication mechanism. Such facilities are particularly important for networking applications and with this in mind UNIX system V and Berkeley UNIX 4.2 BSD, both due for release in the middle of 1983, have provisions for interprocess communication.

2. UNIX at CERN

In 1981, CERN acquired a UNIX source licence and introduced an experimental UNIX V7 service based on a PDP-11/45. The aim of the service was to evaluate the role that UNIX could play in the areas of software development and document preparation.

In terms of hardware the PDP-11/45 has the maximum 256 kbytes of memory and several small disc packs which, in total, provide about 30 Mbytes of disc space. Of the 30 Mbytes, about 18 Mbytes are available as user file space, the rest being used for the system and on-line documentation and manuals. A CAMAC crate is also connected to the machine to provide the hardware connection to the main CERN network, CERNET [7].

The current service based on the PDP-11/45 is severely limited both by the 16-bit addressing of PDP-11 machines and by the physical memory limitation of 256 kbytes on the model 45. When, as a result of the evaluation, it was decided to base future microprocessor support on UNIX systems, we agreed to set up a full-scale service based on a VAX 11/780 system running Berkeley UNIX.

2.1. Connection of the UNIX PDP-11/45 to CERNET

CERNET is a packet-switching network and is primarily used to link small or medium minicomputers with the central mainframe machines, IBM 3081, Siemens 7880 and CDC 7600. These mainframes offer large file

bases as well as access to specialized peripherals such as laser printers and a mass store. User services in CERNET are based on a number of protocols that are arranged in a series of layers. This "onion skin" approach is similar to that proposed in the open system model of the International Organization for Standardization but, in some cases, the exact functions performed by the protocol layers differ.

(1) *The line level protocol* handles the correct transmission of packets between adjacent nodes in the network. In CERNET these packets are at most 2046 bytes in length.

(2) *The transport service protocol* layer is responsible for the establishment and maintenance of the virtual circuit which is used by any two processes that wish to communicate over CERNET. The transport layer treats such problems as flow control, message decomposition into packets and their eventual reassembly, packet routing etc. Implementation of the transport layer in an efficient manner involves a considerable software effort and this has some bearing on the manner in which the UNIX PDP-11/45 is connected to CERNET.

(3) The most important of *the application layer protocols* in CERNET is the file access protocol. This provides remote file access at the record level and is heavily used for file transfer and archiving to the IBM central file base.

The absence of general interprocess communication facilities in UNIX implies that a full CERNET transport service cannot easily be implemented in an efficient manner. Thus, it was decided to connect the PDP-11/45 to CERNET using the simple file transfer (SFT) approach [8]. The SFT protocol was originally developed for bootstrapping small machines using software in the read-only memory and therefore it offers only a limited subset of the CERNET file access facilities. However, it has the great advantage of not requiring a full transport service in the subscribing machine. Transport layer services are managed by software in the nearest CERNET node. SFT does not allow file access at the record level but, instead, the complete file must be transferred to the subscribing machine. In practice, this limitation has not been a serious restriction for the UNIX connection to CERNET.

3. Input–output under UNIX

This section is a brief review of the UNIX I/O system and its relation to the implementation of the CAMAC driver. Much of what is mentioned here is described in greater detail in two excellent articles in the *UNIX Programmers' Manual* [9] and the *Bell System Technical Journal* [10].

UNIX distinguishes between two categories of device: "block" devices and "character" devices. Block devices have a relatively complex interface to the system and are used for highly structured devices such as discs and tapes, typically devices which are capable of supporting a file system. Access to a block device is always through a layer of intermediate buffering software which operates using a cache of system buffers. This approach can

reduce the amount of physical I/O carried out by the system but the latency in the final output to the device can give rise to problems in the event of a system crash.

Character devices have a simpler system interface and are used for all other devices including our CAMAC system. Both block and character devices have an associated "major" and "minor" device number. The major device number identifies the type of device and hence the device driver whereas the minor device number is used by the driver to identify a sub-device. As an example, the minor number will be used to identify a particular drive for a disc controller whereas, for a communication device, it is often used to identify a particular virtual circuit.

3.1. UNIX device drivers

Device drivers in UNIX form part of the kernel and are coded in the language C. Access to the device from a process is via one of five interface routines: OPEN, CLOSE, READ, WRITE, IOCTL. The last routine IOCTL is used to pass I/O control functions to the driver. In addition to the five entry points, an interrupt handler must also be specified.

The link between the UNIX kernel and the interface routines for a device driver is a file called the configuration file. This file is a "switch table" and consists of a series of entries, ordered by the major device number, which are the entry points of the five driver routines mentioned above.

4. CAMAC hardware

CAMAC hardware on the CERN UNIX system consists of a Fisher controls system crate with a Unibus interface. Access to the crate from software is via Unibus registers. In addition to two modules for CERNET usage (the sender and receiver) the system crate contains an interrupt vector generator and a look-at-me (LAM) grader. No direct memory access module is included in the hardware and therefore all data are transferred under program control.

With the simplified protocol used, only CERNET receiver LAMs are enabled. The PDP-11 is interrupted whenever a control word or a block of input data has arrived over the physical data link.

5. Software for CAMAC handling

The device driver for CAMAC in UNIX is written entirely in language C and implements CAMAC as a character device. As the driver provides no possibility for the queuing of I/O requests from processes, the CAMAC can only be used by one process at a time. The reservation and release of the CAMAC device are assured by a primitive locking mechanism, similar to that used with files. In addition, each request from the subscribing process is

treated entirely sequentially. These limitations apply to most character devices in UNIX but block devices such as the disc driver are more sophisticated and can service several processes concurrently.

The implementation of the CAMAC software was done in two steps as described in Sections 5.1 and 5.2.

5.1. Implementation of ESONE CAMAC calls

First a minimal subset of European Standards of Nuclear Electronics (ESONE) CAMAC calls [11] was implemented. The OPEN system call assures reservation of the CAMAC device and the CLOSE system call serves to release the device for use by another process. In the event that a process terminates abnormally and is unable to issue the CLOSE call, the UNIX system assumes responsibility for device and file closure.

The ESONE calls are implemented using the IOCTL system call to establish communication between the user process and the driver. In the subset of ESONE calls implemented, no block transfer functions were provided and so it did not make sense to use the READ and WRITE entry points in this first implementation of the software. Some examples of the code for the ESONE call implementation in the driver are given in Appendix A.

5.2. Implementation of the file transfer protocol

The second stage was the implementation of the SFT protocol. Here a primary goal was to follow the UNIX philosophy and to make the remote file access mechanisms as transparent as possible from the user's point of view. Thus, remote file access and local file access should be highly similar operations in the sense that OPEN, READ, WRITE and CLOSE should work for remote files as for local files.

A second driver was written to implement these standard system calls. For the OPEN system call it was not possible to map fully local and remote system calls as OPEN requires an additional parameter for remote file access. In addition, for efficiency, the implementation of the READ and WRITE routines in the driver was not done using ESONE calls but with a tight block transfer loop.

If remote access to a single host file base is restricted, in our case an IBM machine in the CERN computer centre, then the minimum requirement in the OPEN system call is to add a remote file name in addition to the local device name. Thus, a routine IBMOPEN was written that accepts a remote file name as its single input parameter. This routine can then be used to replace the standard OPEN system call in any utility programs where remote file access rather than local file access is required. Using this idea, a series of utilities was developed in the form of UNIX commands to perform the following functions: (1) file transfer between UNIX and the IBM file base; (2) transfer of UNIX print files to the IBM laser printer; (3) file archiving on the IBM using the UNIX "tar" (tape archive) command in conjunction with file transfer commands; (4) incremental dumping of the UNIX file base to the IBM; (5) IBM job control and enquiry.

6. Experience and conclusions

Several lessons were learnt from our experience of connecting CAMAC to a UNIX system.

(1) The addition of a new device driver into UNIX is greatly facilitated by the simple and well-defined interface to the UNIX kernel.

(2) The use of the language C for device drivers allows control structures to be coded in an efficient and yet readable manner. This is in contrast with many systems where device drivers must be coded in assembler.

(3) The debug cycle is quite rapid since rebuilding the UNIX kernel to incorporate a new version of a driver takes only a few minutes.

(4) As regards the debug tools themselves, the UNIX debugger is terse and cryptic but, nevertheless, quite a powerful tool.

(5) The limitations of UNIX in the area of real-time applications have not been a serious problem for our particular application of connecting to CERNET via CAMAC. Moreover, it should be noted that several of the UNIX limitations listed in Section 1.5 will disappear with release of system V and Berkeley UNIX 4.2 BSD.

Acknowledgments

We would like to thank Peter Villemoes and Kate Spence for their invaluable help and the major contribution that they made in integrating the SFT facilities as commands in the UNIX system.

References

1 D. M. Ritchie and K. Thompson, The UNIX time-sharing system, *Commun. ACM, 17* (7) (1974) 365 - 375.
2 *Bell Syst. Tech. J., 57* (6) (1978) 1897 - 2312.
3 S. R. Bourne, *The UNIX System*, Addison-Wesley, Reading, MA, 1982.
4 B. Kernighan and D. M. Ritchie, *The C Programming Language*, Prentice-Hall, Englewood Cliffs, NJ, 1978.
5 H. Lycklama and D. L. Bayer, The MERT operating system, *Bell Syst. Tech. J., 57* (6) (1978) 2049 - 2086.
6 D. M. Harland, Running a realtime process and a time-shared system (UNIX) concurrently, *Software — Pract. Exper., 10* (1980) 273 - 281.
7 J. M. Gerard (ed.), CERNET — a high-speed packet-switching network, *CERN Yellow Rep. 81-12*, 1981 (Organisation Européenne pour la Recherche Nucléaire).
8 D. Brown and C. Piney, Bootstrapping and simplified file transfer, *CERNET Project Note 88*, 1981 (Organisation Européenne pour la Recherche Nucléaire).
9 D. Ritchie, The UNIX I/O system, *UNIX Programmers' Manual*, Vol. 2b, 7th edn., 1979, Section 32.
10 K. Thompson, UNIX implementation, *Bell Syst. Tech. J., 57* (6) (1978) 1931 - 1946.
11 Subroutines for CAMAC, *ESONE Doc. SR/1*, 1978 (European Standards of Nuclear Electronics).

Appendix A

A.1. Examples of C programming in the CAMAC driver

The following essential data structures are used in the ESONE call implementation.

```
struct cssab {                  /*control block used for cssa and cfsa  */
    int ext;                    /*the coded b, c, n, a combination      */
    char top;                   /*the top 8 bits of a 24-bit value      */
    unsigned q:1;               /*a 1-bit q value                       */
    unsigned f:7;               /*a 7-bit function value                */
    int ints;                   /*the lower 16 bits of a 24-bit value   */
};
struct cdregb {                 /*control block used for cdreg          */
    int ext;                    /*result of cdreg: coded b, c, n, a     */
    char b;                     /*branch number                         */
    char c;                     /*crate number                          */
    char n;                     /*station number                        */
    char a;                     /*register number                       */
};
struct device {                 /*control block desc Unibus registers   */
    int cmcsr;
    char cmdbh;
    char cmnotusd;
    int cmihr;
    int cmggl;
};
```

Other important definitions are as follows.

```
#define CAMAC ((struct device*)0166000)     /*defines Unibus address */
#define FIELD0          0160000
#define CMBCMASK        077000
#define CMANMASK        0777
#define CMWBIT          020
#define CMTBIT          010
```

The following example of ESONE CAMAC call implementations in the driver shows the essentials of routines cmcdreg and cmcssa.

```
cmcdreg(cmp)
    register struct cdregb *cmp;
    {
    cmp->ext = ((int)(cmp->b)<<12)|((int)(cmp->c)<<9)|((int)(cmp->a)
                <<5)|(cmp->n);              /*BBBBCCCAAAANNNNN */
```

Cmcdreg is called with the address of an appropriate control block structure (as defined above) as an argument. It then picks out the b, c, n and a numbers, converts them to integers and shifts them into place. These data are then ORed together and the result is stored back in the control block.

```
cmcssa(cmp)
    register struct cssab *cmp;     /* arg is addr(cssa control block)          */
    {
    register struct device *camp; /* structure to address Unibus regs           */
    int *cur;                       /* Unibus address used in read/write/test */
    int csr;                        /* local variable to construct csr contents */
    camp = CAMAC;                   /* set local pointer to Unibus address(csr)*/
    /* set csr from (b, c, f) and set Unibus address from (a, n)                */
    csr = (((cmp->ext)&CMBCMASK)>>1)|(cmp->f);
    cur = (((cmp->ext)&CMANMASK)<<1)|FIELD0;
    camp->cmcsr = csr;                          /* set csr                      */
    if ((cmp->f)&CMTBIT)                        /* test operation               */
            { if (*cur); }                      /* do nothing (but set q)       */
    else  {
          if ((cmp->f)&CMWBIT)                  /* write operation              */
             *cur = cmp->ints;                  /* write 16-bit value           */
          else                                  /* read operation               */
             cmp->ints = *cur;                  /* read 16-bit value            */
    }
        cmp->q = camp->cmcsr<0?1:0;             /* if csr<0 then 1 else 0       */
}
```

As an example of the very concise coding that is possible in language C, let us look at the loop that is used to output a block of data to a CAMAC register. Let us assume that the block start address is held in variable wd and the word count in variable wcount. Also let us assume that the Unibus address to be used in the transfer is in variable cur and that variable camp holds a pointer to the Unibus address of the CSR. The code to implement block data transfer in Q-stop mode then reads as follows.

```
        camp->cmcsr = 16;                   /* set function value in csr    */
        do   {
             *cur = *wd++;                  /* write 16 bit, incr. pointer */
             if (camp->cmcsr> = 0) break;   /* stop if q = 0                */
             } while (--wcount);            /* dec wcount, stop if zero     */
```

As can be seen from this example, an efficient C compiler can generate code that is as efficient as programming in assembler.