XEROX

Internal Memo

To	From
Ted Strollo	Greg Thomas
Subject	Date
Ethernet Software	June 15, 1979

The accompanying disk contains an RT-11 Operating System along with all of the source (BCPL and MACRO11), binary, load, and command files required for the operation and maintenance of the Ethernet software.

The PDP-11 Ethernet software is composed of the following Alto packages:

EFTP Protocol & Program
PUP Levels 1 & 0
Context
Queue
Alloc
Timer

All Alto BCPL code was syntactically modified to conform to the requirements and limitations of the PDP-11 DOS compiler, but is otherwise unchanged. All assembly language code was rewritten in MACRO11. Because the code is virtually unchanged from the Alto implementation, the Alto documentation is totally definitive and trustworthy; however because of a PDP-11 BCPL limitation, defaulted arguments must be set to zero (rather than omitted). The documentation for the assembly language interface to BCPL routines is attached to this letter along with a definition of the NDB which has been changed for compatibility with the Ethernet hardware. All files of the form *.PAL are MACRO11 code which was generated by the BCPL compiler running under the DOS Operating System, and all files of the form *.MAC were hand coded in MACRO11.

All code is independent of the operating system except for EFTP Program and a routine called OSA.MAC. EFTP Program uses monitor functions (contained in RT11.MAC) to read/write the operator's console and the system disk. OSA.MAC sets up the stacks, initializes the timer, and allocates memory for the use of the Alloc package; it also contains routines which were a part of the Alto Operating System (MoveBlock, Zero, Noop, SetBlock, CallSwat, SysErr, Usc, GetFixed, and FixedLeft).

Greg Thomas
/alm

Basic System Notes

1. Store Allocation

The code of compiled BCPL programs and libraries resides in high store, immediately above the BCPL global vector. The PDP system stack, which is only allocated enough space to perform monitor tasks, is below the global vector and below the system stack is the BCPL runtime stack, which runs down store.

```
        HI  store       |-------|  <----
                        |       |      |
                        |       |      |
                        | code  |   linker
                        |       |   output
                        |=======|      |
                        |globals|      |
                        |=======|  <----
                        |  SP   |
                        |=======|
                        |       |
                        |       |
                        | BCPL  |
                        | stack |
                        |       |
                        |       |
                        |.......|  <- top of DOS buffers
                        |       |
                        |       |
                        |=======|  <- top of resident DOS
                        |       |
                        |  DOS  |
                        |       |
        LO  store       |-------|
```

The DEC DOS monitor lives at the bottom of store and acquires transient space from the store immediately above itself. The BCPL I/O library also obtains space for stream buffers from this area, which is administered dynamically by DOS.

2. Register Allocation

The BCPL stack register is general register zero, the system stack register (the SP) and the program counter (the PC) are necesarily registers six and seven.

On function entry registers one to four are used to pass the first four arguments, on function return register one holds any result. The only use of the system stack by the BCPL system is on function entry to hold temporarily the return link.

3. BCPL Stack Arrangement

As noted the runtime stack grows down store, and is allocated as shown.

```
    HI  store                                          LO  store
                                                  stack ptr
                                                       |
     --------------------------------------------------|-----------------
                   | | |    old frame    | | |  current frame
     --------------^-^---------------------^-^----------------------------
                   | |                     | |
              debug|                  debug|
                     |                       |
               previous                current
               routine                 routine
                link                    link
```

The 'savespace-size' holding the static links of function entry is of size two, one of which is used for the code address linking, and hence also the previous frame size, and the other for debugging information or for use with the Intcode Interpreter.

Vectors are arranged to run up store, according to the BCPL definition. However the "vector" of arguments to a routine does not follow the definition - it grows down store!

4. Global Vector Linking

The Global Vector is known at link time as the Named Csect 'GLOBAL', linking of BCPL programs with this Csect is automatic. At the machine code level the conventional mechanism of accessing this Csect is by assigning a variable G to the address of global zero and offsetting from this address. Thus:-

```
        .CSECT GLOBAL              ;enter Csect Global
               G=.                 ;G = address of global zero

               .=G+101.+101.       ;at global one hundred and one
               FUNC                ;insert the value FUNC
```

The variable G must only be assigned to once per assembler segment.

5. Function calling Sequence

```
                .
                .
        JSR     PC,@G+N            ;calling through global N/2
                M                  ;frame size M+2 bytes
                .
                .
```

6. Function Entry Sequence

```
        SUB     @0(SP),R0       ;standard entry code
        MOV     (SP)+,-(R0)     ;end of entry sequence
        MOV     R0,R5           ;copy known args to the stack
        MOV     R1,-(R5)        ;first arg on
        MOV     R2,-(R5)        ;second on, etc up to four args
                .
                .               ;code of the routine
```

7. Function Exit Sequence

```
                .               ;code of the routine
                .
        MOV     (R0)+,R5        ;result, if any, must be in R1
        ADD     (R5)+,R0
        JMP     (R5)            ;return completed
```

8. Code for Debugging Aids

On function entry the address of the called routine can be saved on the runtime stack in the link; the entry code then becomes,

```
        .BYTE  7 ,'A'           ;a BCPL string which is
        .BYTE 'B','C'           ;the name of the function.
        .BYTE 'D','E'           ;the string, if present, is
        .BYTE 'F','G'           ;always of length seven chars.
        SUB     @0(SP),R0       ;as normal
        MOV     PC,(R0)         ;save the PC on the stack
        MOV     (SP)+,-(R0)     ;as normal
                .
                .               ;etc
```

Profile counting is performed by the sequence

```
                .
                .
        INC     (PC)+           ;add one to the current count
                0               ;the current count
                .
                .
```

Further facilities are under development. (e.g. trace routines)

9. BCPL Addresses

At all times it must be remembered that BCPL manipulates addresses as integers. These integers are the addresses of consecutive sixteen bit fields in store and hence must be word addresses. To convert a BCPL address to a machine address one must thus convert to a byte address, which is most easily performed by a single left shift.

10. BCPL Strings

BCPL strings are vectors, considered as a sequence of bytes, the less significant half of each word preceeding the more significant, and these pairs being treated in their order of appearance in the vector. The value of the first byte of the string is the number of bytes in the string, excluding itself.

May 1974
SRL

Data Structures

Queue Structures

The Ethernet Software makes extensive use of queues. PBI's exist on any of three queues: an input queue (pbiIQ), an output queue (oQ), and a free queue (pbiFreeQ). There is also a queue of Network Data Block's (NDB); these exist on ndbQ. At the present time the only NDB is for the Ethernet (EtherNDB). Another queue in the system is the packet filter queue (pfQ); it is a queue of the names of programs which are to be used in determining the validity of received packets. The queue heads for oQ and pfQ are located within an NDB. The structure of a queue is a follows:

Queue Head

head	pointer to first item in queue (0 if empty)
tail	pointer to last item in queue (0 if empty)

Queue Item

link	pointer to next item in queue (0 if last item)

EtherNDB Structure

The EtherNDB contains all the information necessary for the operation of the Ethernet and its hardware:

EtherNDB

link	pointer to next NDB	
localNet	local net number - zero if unknown	
localHost	local host number	
netType/deviceNum	type of network/address of hardware	
numGPBI	PBI's allowed to gateway for this net	
pfQ: head	queue of packet filters	- first item
tail		- last item
pupPF: link	PUP packet filter	- pointer to next filter
predicate		- address of EtherPupFilter
queue		- address of pbiIQ
encapsulatePup	address of EncapsulateEtherPup	
level0Transmit	address of SendEtherPacket	
level0Stats	address of SendEtherStats (reserved)	
iCommand: count	2's comp of input word count	
address	input buffer location	
status	input control & status word	
iPBI	input PBI being processed	
oCommand: count	2's comp of output word count	
address	output buffer location	
status	output control & status word	
delay	random number for output delay	
load	load mask for random countdown	
xmtTimeout	transmitter timeout	
oPBI	output PBI being processed	
oQ: head	output queue	- first item
tail		- last item

PBI Structure

A PBI is a buffer that contains a PUP and information pertaining to it.

PBI

link	pointer to next PBI	
queue	address of xmitted-PBI queue	
socket	address of owning socket	
ndb	address of NDB for this PBI	
status	PBI control info	
timer	retransmission timer	
packetLength	no. of words in packet	
dest/src	dest host / src host	
type	packet type	
length	length of PUP (bytes)	\
transport/type	PUP control / PUP type	|
id(1)	sequence no. (1)	|
id(2)	sequence no. (2)	|
dPort: net/host	dest net / host	|
socket(1)	dest socket id(1)	|----- PUP
socket(2)	dest socket id(2)	|
sPort: net/host	src net / host	|
socket(1)	src socket id(1)	|
socket(2)	src socket id(2)	|
words	data words in PUP	/

Context Structure

A context exists on a context queue and is used by the context package to control the execution of tasks on a round-robin basis. The context contains a stack pointer, some space reserved for the user, and a stack (which contains a resume execution address).

Context

link	pointer to next Context
sp	current stack pointer
exspac	space reserved for user
stack	stack used by task