Remote

Designing a Remote System Program

by Al J. Szymanski

The program that I am presenting here came about because of a need on my part to move files from two incompatible systems. This program may also become the core for a fully featured remote system driver. I have a Tiny Giant 68000 with K-OS and I love it. I also have my trusty CP/M system which I use as a development system. Herein lies my first headache, format incompatibility. The K-OS uses the MSDOS/IBM format for disks and I cannot utilize that format with my CP/M system. So I wrote this program to allow me to drive the K-OS system as a remote from the CP/M system. Specifically, I now can develop code on the Z80 and then ship it to the 68000 to assemble and run (More on WHY later). Additionally, in order to send TCJ the code and articles on one disk I had to be able to ship code from the 68000 back to the Z80. I generally try to write straightforward code in the sense of making tools, if it works for me then it's OK. This is the first effort I've made to make code as bombproof and friendly as I could. I also realize that we all hate to re-invent the wheel, thus the reason for my sharing this code.

I will discuss each part of the program, as it appears in the listing. HTPL is a Forth-like programming language that has been covered in previous articles. It uses the stack for the evaluation of variables. Since it is very sensitive to any garbage left on the stack, your code must be very clean. I felt that it was fairly easy to create simple tools with HTPL, also it was the only language available on the 68000 system. As I am a die-hard C programmer, it is my intent to port C over to the K-OS environment.

First up are the variable declarations, **AUX:** being the filename of the auxillary port. **Bbuf** is the string input buffer, **Rwbuf** is a buffer for reading and writing to and from the disk. **Abuf** is a small character buffer. **Rec1** and **rec2** are the received record number and its complement, used to verify logical sequencing of the records. **Try** is the counter for the

```
(   1)  ( remote. v1.0 in HTPL by Al J. Szymanski 11/87)
(   2)  root
(   3)  byte menu = "MENU: R receive, S send, V view, Q quit :>" ;
(   4)  byte smsg = "Send " ;
(   5)  byte rmsg = "Receive " ;
(   6)  byte vmsg = "View " ;
(   7)  byte cmsg = "Command not implemented." ;
(   8)  byte qmsg = "Quitting Remote." ;
(   9)  byte fmsg = "Filename.ext ? " ;
(  10)  byte auxname = "AUX:" ;
(  11)  byte bbuf [ 128 ] ;
(  12)  byte rwbuf [ 512 ] ;
(  13)  byte abuf [ 4 ] ;
(  14)  byte rec1 rec2 try response checksum ;
(  15)  word lastrec status filchan auxchan ;
(  16)  word time = 2500 ;   (this is a MAGIC number, this works for 1200 baud)
(  17)  word pblock [ 20 ] ;
(  18)  long bufptr ;
(  19)
(  20)  program
(  21)       openaux
(  22)       "Remote.V 1.0 - Ready" writeln
(  23)       while auxgetc 3 <> do
(  24)            "Remote.V 1.0 - waiting for  control^C" auxputs 13 auxputc
(  25)       end
(  26)       repeat #menu auxputs auxgetc toupper auxcrlf
(  27)       case     [ 'R' ] receive
(  28)            [ 'S' ] send
(  29)            [ 'V' ] view
(  30)            [ 'Q' ] quit
(  31)            else dontknow
(  32)       end
(  33)       false until
(  34)  end
(  35)
(  36)  (**********PRIMARY ROUTINES**********)
(  37)
(  38)  proc receive
(  39)       #rmsg auxputs getfilec #rwbuf !bufptr 0 !lastrec
(  40)       while auxstat =0 do 21 auxputc @time wait end
(  41)       repeat
(  42)            auxin dup
(  43)            if 1 = then
(  44)                 6 !response
(  45)                 auxgetc !rec1 auxgetc not !rec2
(  46)                         0 dup !checksum
(  47)                 while dup 128 <> do
(  48)                      dup auxgetc dup @checksum + !checksum
(  49)                      @bufptr rot + 1! +1
(  50)                 end drop (128)
(  51)                 @checksum $00FF and auxgetc $00FF and
(  52)                 if <> then 21 !response end
(  53)                 @rec1 @rec2
(  54)                 if <> then 21 !response end
(  55)                 @response 6 if = then awrite end
(  56)                 @status $0004 and $0004
(  57)                 if = then 24 !response end
(  58)                 @lastrec +1 $00FF and @rec1 $00FF and
(  59)                 if <> then 24 !response end
(  60)                 @rec1 $00FF and !lastrec
(  61)                 @response auxputc
(  62)                 @response 24 if = then return end
(  63)            end
(  64)       4 = until.
(  65)       6 auxputc aclose auxcrlf
(  66)  end
(  67)
(  68)  proc send
(  69)       #smsg auxputs getfileo
(  70)       0 !rec1 while auxgetc 21 <> do end
```

attempts made at sending a record, 5 is the current limit. **Response** is the variable used to hold the response to make after receiving a 128 byte packet. It may be either: NAK or 21d , ACK or 6d , CAN or 24d. These are the codes used in the xmodem or Christensen protocol as handshakes. **Checksum** is the sum of the 128 bytes in the packet mod 256. **Lastrec** is the number of the last record received, used to verify that the record that was just received is the next in order. **Status** holds the contents of the status word from the last operating system call made. **Fichan** and **Auxchan** are the file channels or descriptors for the currently open file and for the auxillary port. K-OS treats all files and devices as channels. The word **time** is a variable that was needed to slow down the process of handshaking. (Probably half of the bugs I encountered while working this code out, were due to incoming bytes being stored in the character queue on both machines. This meant that there were occasionally garbage characters waiting and being interpreted as handshakes. This value came about by testing empirically as opposed to calculation and it works for 1200 baud, I don't know what would work for 300). **Pblock** is the structure for all of the operating system calls. **Bufptr** is a long pointer to a byte in the **wrbuf**.

The next block of code is the core of the program. It proceeds as follows: (line 21) open up the channel to the auxillary port, (line 22) send to the 68000 terminal a message that the program is up and running, it's time to switch over to the CP/M machine, (lines 23-24) wait in a loop until a character comes in, assess it for being a ^C, if it is not, send a message asking for the ^C. This was done to clear out the queue. Next enter a large repeat forever loop (lines 26-33) which sends out the menu and waits for a selection. The case evaluates the choice and branches to a routine, with a default at dontknow to bullet-proof the code. The only way out of the code at this level is to enter a 'Q' to quit the program.

Next up are the 5 primary routines; **receive, send, view, quit** and **dontknow**. The basis for what is going on in **send** and **receive** is best described in the article by Donald Krantz, "Christensen Protocols in C," *DR. DOBBS JOURNAL* (#104 June 1985 pp. 66). It is the clearest presentation of the xmodem protocols I have ever read.

Receive works this way: (line 39) it asks for the filename.ext to create and then does so, then starts sending 21d's (NAK)

```
(  71)      repeat
(  72)           aread128 0 !try @rec1 +1 !rec1
(  73)           repeat
(  74)                1 auxputc 0 dup !checksum
(  75)                @rec1 $00FF and dup auxputc not auxputc
(  76)                while dup 128 <> do
(  77)                     dup #rwbuf + @1 dup
(  78)                     @checksum + !checksum
(  79)                     auxputc +1
(  80)                end drop
(  81)                @checksum $00FF and auxputc
(  82)                @try +1 dup !try 11 5 = then errorout end
(  83)           auxgetc dup 24
(  84)           11 = then drop aclose return end
(  85)           6 = until
(  86)      @status $0008 and <>0 until
(  87)      4 auxputc
(  88)      aclose
(  89) end
(  90)
(  91) proc view
(  92)      #vmsg auxputs get1ileo auxcrlf @time wait 0 !status
(  93)      while @status $0008 and =0 do
(  94)           aread1 dup auxputc
(  95)           if 26 = then
(  96)                aclose auxcrlf return
(  97)           end
(  98)           if auxstat 1 = then
(  99)                auxgetc
( 100)                if 3 = then
( 101)                     aclose auxcrlf return
( 102)                end
( 103)           end
( 104)      end
( 105) end
( 106)
( 107) proc quit
( 108)      #qmsg auxwriteln  exit end
( 109)
( 110) proc dontknow
( 111)      #cmsg auxwriteln end
( 112)
( 113) (**********O.S. CALLS**********)
( 114)
( 115) proc openaux
( 116)      #pblock 5 over 12  0 over +2 12
( 117)      0 over +4 12 #auxname over 6 + 14
( 118)      trap     #pblock +4 @2 !auxchan
( 119) end
( 120)
( 121) proc auxstat
( 122)      #pblock 1 over 12 0 over +2 12
( 123)      @auxchan over 4 + 12
( 124)      trap #pblock 2 + @2 1 and (returns 1 if char waiting else 0)
( 125) end
( 126)
( 127) proc auxin
( 128)      #pblock 2 over 12 0 over +2 12
( 129)      @auxchan over +4 12 1 over 6 + 12
( 130)      0 over 8 + 12 #abuf over 10 + 14
( 131)      trap @abuf      (returns the char on stack)
( 132) end
( 133)
( 134) proc auxputc
( 135)      !abuf #pblock 3 over 12 0 over +2 12
( 136)      @auxchan over +4 12 1 over 6 + 12
( 137)      0 over 8 + 12 #abuf over 10 + 14
( 138)      trap
( 139) end
( 140)
( 141) proc acreate
( 142)      #pblock 6 over 12 0 over +2 12
( 143)      swap over +4 14 0 over 8 + 14
( 144)      trap #pblock +2 @2 !status
( 145) end
( 146)
( 147) proc aopen
( 148)      #pblock 5 over 12 0 over +2 12
( 149)      0 over +4 12 swap over 6 + 14
( 150)      trap #pblock +4 @2 !fichan
( 151)      #pblock +2 @2 !status
( 152) end
( 153)
( 154) proc aclose
( 155)      #pblock 8 over 12 0 over +2 12
( 156)      @fichan over +4 12
( 157)      trap #pblock +2 @2 !status
( 158) end
( 159)
( 160) proc awrite
```

```
( 161)       #pblock 3 over !2 0 over +2 !2
( 162)       @flchan over +4 !2 128 over 6 + !2
( 163)       0 over 8 + !2 @bufptr over 10 + !4
( 164)       trap #pblock +2 @2 !status
( 165)  end
( 166)
( 167)  proc aread1
( 168)       #pblock 2 over !2 0 over +2 !2
( 169)       @flchan over +4 !2 1 over 6 + !2
( 170)       0 over 8 + !2 #rwbuf over 10 + !4
( 171)       trap #pblock +2 @2 !status
( 172)       #rwbuf @1       (return read char on stack)
( 173)  end
( 174)
( 175)  proc aread128
( 176)       #pblock 2 over !2 0 over +2 !2
( 177)       @flchan over +4 !2 128 over 6 + !2
( 178)       0 over 8 + !2 #rwbuf over 10 + !4
( 179)       trap #pblock +2 @2 !status
( 180)  end
( 181)
( 182)  proc wait   (expects time on stack)
( 183)       #pblock 53 over !2 0 over +2 !2
( 184)       swap over +4 !4 trap
( 185)  end
( 186)
( 187)  (**********UTILITY ROUTINES**********)
( 188)
( 189)  proc returnstat
( 190)       @status
( 191)       if $8000 and $8000 = then
( 192)            "processed " auxputs
( 193)       end
( 194)       @status
( 195)       if $0004 and $0004 = then
( 196)            "unsucessfully." auxputs
( 197)       else "sucessfully." auxputs
( 198)       end
( 199)  end
( 200)
( 201)  proc crlf
( 202)       13 putc 10 putc end
( 203)
( 204)  proc auxcrlf
( 205)       13 auxputc 10 auxputc end
( 206)
( 207)  proc writeln
( 208)       sprint crlf end
( 209)
( 210)  proc auxwriteln
( 211)       auxputs auxcrlf end
( 212)
( 213)  proc auxgetc
( 214)       while auxstat =0 do end auxin end
( 215)
( 216)  proc auxgets
( 217)       while auxgetc dup <>0
( 218)            do
( 219)                 case
( 220)                 [ 17 ] 30 exit
( 221)                 [ 13 ] drop 0 swap !1 return
( 222)                 [ 8 ] auxputc -1 32 auxputc 8 auxputc 0
( 223)                 else dup auxputc over !1 +1 0
( 224)                 end
( 225)       end
( 226)  end
( 227)
( 228)  proc auxputs
( 229)       while dup @1 dup <>0 do auxputc +1 end
( 230)       drop (pointer to end of string)
( 231)  end
( 232)
( 233)  proc errorout
( 234)       24 dup auxputc auxputc exit end
( 235)
( 236)  proc getfileo
( 237)       filine #bbuf dup auxgets auxcrlf aopen returnstat end
( 238)
( 239)  proc getfilec
( 240)       filine #bbuf dup dup auxgets auxcrlf acreate aopen returnstat end
( 241)
( 242)  proc filine
( 243)       #fmsg auxputs end
( 244)
( 245)  proc toupper
( 246)       dup if 96 > then 32 - end end
( 247)
( 248)  end
( 249)  end
```

until any response is made, then it enters a large loop (lines 41-64) which begins by getting the response and evaluating it for being a 1 or SOH (start of heading), a 4 or EOT (end of text), all other responses being treated as junk. If the response is a SOH it sets up its own response to be a 6 or ACK (acknowledge). Then it gets the next incoming byte which should be the logical record followed by the complement of the logical record. It then enters a small loop (lines 47-50) that just gets the next 128 bytes from the stream and puts them into memory, while calculating the checksum for the record. The byte that follows the data is the sent checksum, which should match our calculated one, if not we change our response to NAK (line 52). It then evaluates the sent record and its complement for errors, again changing the response on error (line 54). If by now no change has been made in the response, it writes the 128 bytes to disk. An evaluation is then made to see if the write went OK, if not the response is changed to 24d or CAN (cancel) which puts an immediate end to the data exchange for both machines. Lastly an evaluation is made to be sure that the record we just got was in fact the next record we should have gotten, if it is not, again we set up to abort the exchange. Finally, the response is sent to the sending machine and exiting if it was the CAN byte. If the first byte sent for the next block was the EOT we have reached the end of the loop and can close the new file and return to the menu.

Send is a much simpler routine. As there can be no transmission errors, all that send needs to do is calculate the checksum and send it and the record numbers when it needs to. First (line 69) asks for the filename.ext to send—opens it and reports status. Then (line 70) waits in a loop until a NAK is received. The user must make sure that checksum mode is set on the receiving machine as the sync byte for CRC mode is 'C', and that is ignored by this routine, (this could cause potential lockup). Once the routine has received the sync byte, it enters a large loop (lines 71-86) in which 128 bytes are sent. **Aread128** is an O.S. call which reads 128 bytes into a buffer. The record count is incremented. A smaller loop (lines 73-85) is then entered. This loop first sends the 1 or SOH byte, starting the transmission block. It then clears the checksum. Next it gets the current record, makes sure it is mod 256 and sends it and its complement (line 75). The inside loop (lines 76-80) actually does the transmission of the 128 bytes of data, calculating the checksum along the way.

Fig. 1

```
(address of pblock) top of stack
(value of command )
(address of pblock)
```

Fig. 2

```
address of pblock ==>  | word width |  command value
                       |      3     |
                       |      0     |  status word
                       | file chan. # |  from open
                       |    128     |  number of bytes
                       |      0     |  number written
                       |buffer address of source data|  long data
```

The checksum is then sent, again being corrected for mod 256. The try count is incremented and evaluated against bounds, exiting if it exceeds the limit. The next line (line 83) gets the response from the receiving machine, evaluating it for being 24 or CAN, exiting if it is. This meant that the receiving machine found a non-recoverable error. The routine looks for a 6 or ACK as a response, meaning that the record was accepted correctly. The next line (line 86) checks the status word from this last read for the end of the file, and as long as it is not, returns to the top of the loop. If it were the end of the file, a 4 or EOF is sent to complete the transfer, close the file and return to the menu.

View is even simpler than **Send** as no calculations are made, the data is just sent to the receiver's screen. The code is straightforward, however note the check for ^C in the loop (lines 100-101) to cancel the display of the file. **Quit** and **Dontknow** are unremarkable routines.

The next group of routines are the actual calls made to the K-OS operating system. A parameter block is used to handle all of the pointers and values used in an O.S. call. The first word in the block is the actual command code—usually followed by a status word. One of the beauties of this type of arrangement is that you can make as many parameters blocks as you might need and place them anywhere in memory. Pre-loading of parameter blocks and just issuing the trap call at the time of need can save a great deal of time for critical operations. I'll describe just one of the calls for example sake, **Awrite: #pblock** gets the address of the call buffer and places it on the stack, 3 puts the number 3 onto the stack above the address, 'over' is the HTPL macro word that takes the next to the top item on the stack and makes a copy of it and places the copy onto the top of the stack (see Figure 1). The '!2' means: take the top item—treat it as an address and put the next item down into that address, here (line 161) it means put the value 3 into the address of pblock. In doing this, the top two items are removed from the stack, leaving only the original copy of the address. The code proceeds similarly until the '+2' which adds 2 to the top item on the stack, here (line 161) the address of the pblock, offsetting the pointer by one word. It continues until the '@fichan' which means: put onto the stack the item found in the variable **fichan**. By the time the word 'trap' is reached the stack has only the address of the pblock on it, and trap performs the O.S. call. The

parameter block looks like Figure 2 before the call is made. After the call is made there is nothing left on the stack from this routine, so we have to replace the address of the parameter block onto the stack to get access to the status word. Then we get the word and store it in the variable status.

The final group of routines are the utility routines which allow for the byte by byte exchange through the auxilliary port and with the 68000 screen. Included in this group is the routine **Returnstat**, which evaluates the status word left from the last operation made and displays the information on the host machine. **Auxgets** allows for inline correction through the auxilliary port while getting a string from the host. **Toupper** does have one quirk in that if the characters '{' through '~' are used it will make them unusable, or at best treat them as the control characters ^[through ^~.

I have used this program now for about a month to make a cross compiler to port a version of C onto the 68000 to run under K-OS. There are a few changes that I would suggest be made. One is to allow the host to view the directory of the 68000 machine and eventually give the host full command level capabilities, even to writing a version of BYE for a full remote operating system. As far as my version of C, I have the cross compiler up and running and have most of the 68000 run time library done. I owe credit to the fine folks at Hawthorne Technology. They are only 40 miles up the road from me and have helped me a lot.

I plan to write a few more articles about the Tiny Giant and on the C compiler I'm working on. That's all for now folks. ■

Reader's Feedback

(Continued from page 5)

natural for combining high-level language with various assembler routines.

What have I been doing? Well, a while back, I indicated that I was building a linear supply for a second SB180. It's almost done and I intend to write up an article on how I designed and built it. Perhaps it will be good enough for TCJ to publish.

I've also been doing a bit of PC Board design on the Mac using MacDraw and some templates that I've designed on my own. Be glad to share that experience as well if your readers might be interested.

T.M.

Editor's Note: We'll be looking forward to power supply article, and encourage the readers to let us know about their interest in PC Board design on the Mac.

Hardware Control

I'm using MTU 130, Mac+, Mac SE, Apple IIgs, Apple IIe, and HP Vectra.

Most of the effort is using the above for data collection and some number crunching. I primarily use True Basic and C for programs, and 6502 assembly for some speed in the Apple II.

I would like a good tutorial on 68000 assembly, and also on FORTH (it seems to me to be a nice language but I haven't sat down to learn it). I also really enjoy articles on hardware control, stepper motors, ADCs, DADs, etc.

D.M.

Ripe Thinking

I'm using PCTECH X16B 10MHz with new OMTI 3520 CCS, controller of two different drives, ST 225 and Minis 3650. Earl Hinrichs software is outstanding.

I used to use (before my desert house in 29Palms was burglarized) in addition to the X16B, a CP/M-86/MS DOS environment: FALCO TS1 terminal, Slicer with Shugart 860-2 and two Mitsi 4853s, housed in a Ferguson BB cabinet with Ferguson UPS. Damnation! The first computer I built, ripped off by someone who didn't take the manuals with the system.

Next quarter, I plan to add a TinyGiant 68000. 68000 is the way to go. I don't like DOS or segmented 86. DOS is a real challenge to learn as a first machine, but when you buy computers from PCTECH and Slicer, you get fantastic support that makes the effort worthwhile.

TCJ is more than a breath of fresh air, it's a perspective, e.g. the editorial with emphasis on real time programming.

I'd like to see articles on cross assemblers, like cross assembling 68000 code on the 8086, vice versa, etc. I've been wondering about relatively cheap cross assemblers such as Austin Codework's $25 A68000.

Also interested in new Zilog Z280, Transputers, NS32X32, digital image progessing with NEC uPD7281, and GSP's like TI's 34010.

Concerning the 32X32 and Job's NEXT machine, and other CPUs they're considering, I somethimes think it should be called WHEN Corporation. Facetiousness aside, I am intrigued what the impact of a UNIX machine will have, including the shock of the retail price of the machine.

I think Don Lancaster did hit one nail on the head. when I paraphrase him from "Ask the Guru" in '85/'86: The Macintosh has a fascist operating system — it forces you to be user friendly.

One of the greatest technical BBSs I've used is Trevor Marshalls 1000 Oaks at 805-493-1495.

Turbo C (the only MS DOS C compiler I have, don't know about other ones) has a good feature with the ability to generate symbolic files in command-line version environment that are compatible with MASM's .SYMDEB — those two switches '-y' and '-m' make it really fun to step through executable files.

Saw a demonstration of Tektronix's 3-d color terminal — you put on polarized lenses and a LCD shutter in front of display, and its software makes basic objects (wire frames in this case) pop out of the terminal. Nice toy at $40,000,but like much technology, a matter of time (decade or so) to have a personal 3-D graphic environment, and speaking of that, holographic environments like in debugging — heap's on my left, stack's on my right, registers straight ahead. How long will Von Neumannism survive?

Thanks for TCJ, every issue is for ripe reading/thinking.

R.S.

32-Bit

I use CP/M Z80 S100 and single board, plus UNIX, VAX, MICROVAX, Sun, etc. (college is such fun, eh?).

I would really like to see more hardware projects — especially in the area of 32-Bit single board computers. I am particularly interested in finding out more about the Zilog Z80,000 32-Bit micro-mainframe. How about someone out there making a workstation (Berkeley UNIX based, of course) based on this chip?

R.A.

SB180

I am running a Micromint SB180, with the hardware mods to enable DTR to my Wyse 30 termainal, and four floppies (2 DSDD and 2 DSQD, all TEAC), as well as the 9.216 MHz upgrade and XBIOS. In other words, I have taken my SB180 nearly, but not quite, as far as it will go. Next step SCSI interface and, hopefully, a 2-4 meg RAM disk. (I really don't want to go Hard Drive, though I might end up doing that. Afraid of reliability problems — probably unjustified, however.)

I'd like to see: 1) SCSI, 2) Solid state "drives" for CP/M or Z, 3) Advanced CPU (Z800, Z280, etc), 4) Interface basics — computer with drives, terminals, DMA/keyboard/monitor vs. terminal, modem.

J.B.
