UCSD PASCAL System II.0 User's Manual March 1979

Institute for Information Systems

Introduction

Table of Contents

Table of Contents

1.0 Introduction

1.0 Purpose of This Document

This document is intended to provide a detailed description of the “SYNCHRONOUS” I/O Subsystem of the UCSD Pascal System (An asynchronous I/O Subsystem has yet to be fully defined at this writing). The document is intended primarily for use by persons implementing or maintaining the I/O Subsystem. It is NOT intended to serve as a user's manual for Pascal programmers. 1

Please note that this is only a *preliminary description* of the level 2.0 I/O Subsystem.

1.1 Introduction to the I/O Subsystem

The UCSD Pascal System is constructed in a hierarchical fashion (see Figure 1.0). Most of the system is written and maintained in the Pascal language and is described as the “Pascal Level”. As is discussed elsewhere the UCSD Pascal compiler generates code for an idealized processor known as the *pseudo-machine*. This code (known as *pseudo-code* or p-code) is interpreted at runtime by an assembly language program (known as the *Interpreter*) which emulates the pseudo-machine. Due to the processor-independent nature of the p-codes it is possible to port the entire UCSD Pascal System to a new host machine by rewriting only the Interpreter. Besides emulating the pseudo-machine, native code is also used for some time-critical functions and for dealing with such machine dependencies as input/output devices. The body of code which implements these non-emulatory functions is called the *Runtime Support Package* (RSP). The portion of RSP responsible for communicating with I/O devices is known as RSP/IO.

Due to limited resources, it is our desire to maintain no more than one version of the interpreter for each processor. At the same time we face wide variations in the peripherals which may be encountered. Therefore, we at UCSD sought a scheme by which the code in RSP could be frozen while still allowing for different peripherals. The structure which we have devised is conceptually similar to that used by Digital Research in their CP/M operating system For 0080's and Z-80's. That is. a standard interface has been defined between the configuration- independent RSP/IO code and a configuration- specific *Basic Input/Output Subsystem* (called BIOS) which performs the actual I/O. The semantics of a call to BIOS have been clearly defined as far as what the Pascal level needs to sees but BIOS is allowed to keep all sorts of hairy details (*e.g.* track and sector numbers, sector interleaving) to itself.

Thus we have the *UCSD Pascal I/O Hierarchy* shown in figure 1.0: The Pascal user's I/O calls (*e.g.* WRITELN, READLN, GET and PUT) are mapped bq the Pascal compiler and operating sgstem into calls on RSP (*i.e.* UNITREAD, UNITWRITE). RSP/IO itself calls BIOS which controls the actual device operations. It is important for the reader to recognize that we are here discussing a SYNCHRONOUS I/O system. In other words when an I/O request has been initiated by a Pascal program, control does not return to that program until the I/O operation is completed. It is anticipated that, eventually, a similar I/O system will be specified with asynchronous or “immediate return” capabilities, probably based on the asynchronous system now in use with UCSD Pascal on the PDP-11. 2

3

```
"Pascal level"                  PASCAL USER
                                      |
                                      V
                                PASCAL SYSTEM
                                      |
- - - - - - - - - - - - - - - - - - - | - - - - - - - - - - - - - - - - - -
"Interpreter Level"                   |
(Run-time Support Package)            | unit no., data area address,
                                      | byte count
                                      | [, block no., control word]
                                      V
                                   UNIT I/O
                               (param checking)
                                      |
                                      V
         ,--------------------------------------------------------.
         |                    |                 |                 |
         |Console             |Printer          |Disk             |Remote
         V                    V                 |                 V
   SPECIAL CHAR         SPECIAL CHAR            |            SPECIAL CHAR
     HANDLING             HANDLING              |drive no.,    HANDLING
   write|   |read             |                 |data area        |
        |   |                 |                 |addresss         |
  single|   |single           |single           |byte count       |single
    data|   |data             |data             |logical          |data
    byte|   |byte             |byte             |block no.,       |byte
        |   |                 |                 |control word     |
- - - - | - | - - - - - - - - | - - - - - - - - | - - - - - - - - | - - - -
"BIOS   |   |                 |                 |                 |
Level"  |   |                 V                 V                 V
        |   |              PRINTER             DISK          SERIAL LINE
        |   |            PRIMITIVES         FORMATTER        PRIMITIVES
     ,--'   `------.                      (Map logical
     |             |                        blocks into
     |             V                     track and sector)
     |        TYPE-AHEAD                        |
     |           QUEUE                          V
     |             |                           DISK
     |             V                        PRIMITIVES
     |        SPECIAL CHAR
     |          HANDLING
     |        (start/stop, alpha lock, flush, break)
     |             |
     V             V
  SCREEN       KEYBOARD
PRIMITIVES    PRIMITIVES
```

Figure 1.0 — Pascal I/O System Hierarchy

This page last regenerated Sun Jul 25 01:09:10 2010.

2.0 The Pascal Level

As mentioned above, all Pascal level I/O requests are eventually mapped by the compiler and operating system into calls on a group of UCSD pre-declared procedures known as the *unit I/O procedures*. The Pascal programmer may call the Unit I/O procedures directly or he may use standard Pascal I/O procedures such as `READ`, `WRITE`, `GET` and `PUT`. The exact details of how this mapping is accomplished do not concern us here. The Unit I/O procedures are not written in Pascal but, in facts are the native code procedures comprising the I/O section of the Run-Time Support Package. The mechanism by which they are called is described next.

2.1 Calling the Next Level Down: RSP/IO

All native code routines in RSP are called using the CSP (Call Standard Procedure) opcode, followed in the P-code stream by an unsigned byte containing the procedure number. To the Pascal user making direct calls to Unit I/O routines, they look like any other pre-declared procedure. If they actually were declared in Pascal, the declarations would have the following format (allowing a few illegitimate constructs such as optional parameters and variable length arrays).

```
PROCEDURE UNITREAD(UNITNUMBER: INTEGER;
                   VAR DATAAREA: PACKED ARRAY [0..BYTESTOTRANSFER-1]
                     OF 0..255;
                   BYTESTOTRANSFER: INTEGERS;
                   [ LOGICALELOCK: INTEGER;
                   [ CONTROL: INTEGER ]] );

PROCEDURE UNITWRITE(UNITNUMBER: INTEGER;
                   VAR DATAAREA: PACKED ARRAY [0..BYTESTOTRANSFER-1]
                     OF 0..255;
                   BYTESTOTRANSFER: INTEGERS;
                   [ LOGICALELOCK: INTEGER;
                   [ CONTROL: INTEGER ]] );

FUNCTION UNITBUSY(UNITNUMBER: INTEGER): BOOLEAN;

PROCEDURE UNITWAIT(UNITNUMBER: INTEGER);

PROCEDURE UNITCLEAR(UNITNUMBER: INTEGER; UINITPTR: ^UIR);
```

(Note that UINITPTR is being introduced for the level 2 release.)

Remember that no such declarations actually exist in the system. They are intended to model the parameters passed and returned by the native code RSP/IO routines. Some of these routines are useful only in an asynchronous environment; under the synchronous system described here they are mere dummies.

2.1.1 Units and UNITNUMBERS

The various physical devices of the UCSD Pascal System are numbered, a fixed number being assigned to each device which the system is designed to handle. The formal parameter UNITNUMBER in the declarations above determines which of the Pascal physical units the operation is intended for. Thus the Unit I/O procedures are device-transparent to the Pascal programmer — the same procedure will deal with any of the physical units. Figure 2.0 is a list of the unit numbers associated with each physical unit. The meaning of the other parameters is discussed in section 3 of this document, the section dealing specifically with RSP/IO.

Unit Number	**Volume Name**
0	(must not be used)
1	CONSOLE:
2	SYSTERM:
3	(no current assignment)
4	disk O
5	disk 1
6	PRINTER:
7	(no current assignment)
8	REMOTE:
9	disk 2
10	disk 3
11	disk 4
12	disk 5

Figure 2.0 — Unit Numbers

2.1.2 CONTROL Parameter

The CONTROL parameter to UNITREAD and UNITWRITE is a word used to pass special information to RSP/IO and BIOS regarding the handling of the I/O request. The format of the CONTROL word is shown in Figure 2.1.

```
  15          4           3          2          1          0
+--------------+----------+----------+----------+----------+
| (un-assigned) |   CRLF   |  NOSPEC  | PHYSSECT |   ASYNC  |
+--------------+----------+----------+----------+----------+
  MSB                                                   LSB
```

Unassigned	(3..15)	The contents of bits 4..15 are ignored.
CRLF	(bit 2)	If set, do not do CR LF mapping.
NOSPEC	(bit 2)	If set, implies “no special character handling”, *i.e.* no DLE expansion or LFs appended to CRs.
PHYSSECT	(bit 1)	If set, implies “Physical Sector Mode” for disk I/O.
ASYNC	(bit 0)	Ignored in this implementation.

Figure 2. — CONTROL word format

2.2 IORESULT and Completion Codes

At times, an I/O request will terminate abnormally. To detect such occurrences, UCSD Pascal offers the predefined function IORESULT. IORESULT returns an integer value describing the status of the last I/O request. The value of IORESULT is set as follows: Each call to UNITREAD, UNITWRITE or UNITCLEAR will cause a “completion code” to be set in the SYSCOM data area [SYSCOM (for SYStem COMmunication area) is the one and only data space directly accessible by both the Pascal Operating System and the Interpreter]. Programmers may test the completion code by using IORESULT.

The standard completion codes for release level 2.0 of the UCSD Pascal System are given in figure 2.2.

Code	**Meaning**
0	No error
1	CRC error
2	Illegal unit number
3	Illegal operation on unit
4	*(no longer used)*
5	Lost unit; unit no longer on line
6	Lost file; file name no longer in directory
7	Illegal file name
8	No room; insufficient space on disk
9	Unit not on line; no such volume on line
10	No file; no such file name in directory
11	Duplicate file
12	Not closed; attempt to open an open file
13	Not open; attempt to access a closed file
14	Bad format; error reading real or integer
15	Ring Buffer Overflow
16	Write protect; write attempt to protected disk
17	Illegal block number
18	Non-zero byte count (in physical sector mode)
19	Invalid UIR settings
100..199	Codes 100 through 199 are reserved for non-predefined device-dependent errors.

Figure 2.2 — I/O Completion Codes (Level 2.0)

2.3 Logical Disk Structure

The UCSD Pascal system views the disk as a zero-based linear array of 512 byte *logical blocks*. All UCSD Pascal disks have this logical structure, regardless of their physical format. The physical allocation units of a disk are commonly known as *sectors* and vary widely in size depending on the hardware. The BIOS is responsible for mapping the logical structure of a Pascal disk onto the physical structure of the device, *i.e.* mapping logical blocks onto physical sectors (see section 4.5.3.1).

2.3.1 Physical Sector Mode

8

To provide enhanced flexibility for systems programming at a machine-specific level, a mechanism has been provided for directly accessing the physical sectors of the disk. When the PHYSSECT bit of the CONTROL word is set on a call to UNITREAD or UNITWRITE involving a disk unit. the I/O is performed in *physical sector mode*. This has the following effects:

1. The parameter LOGICALBLOCK is interpreted by the BIOS as *physical sector number* (PSN).
2. The parameter BYTESTOTRANSFER must be zero or an error will be flagged. The actual number of bytes transferred is equal to the physical sector size.

2.3.1.1 Physical Sector Numbers

Typically, the physical sectors of a disk are addressed by specifying both track and sector numbers. That is, the disk is viewed as an *array of tracks* where each track is an *array of sectors*. If this data structure were declared in Pascal. it would look like this:

```
type
     BYTE = 0..255;
     SECTOR = array [0..(BYTES_PER_SECTOR-1)] of BYTE;
     TRACK = array [1..SECTORS_PER_TRACK] of SECTOR;
     DISK = array [O..(TRACKS_PER_DISK-1)] of TRACK;
```

(Note that we are using the convention that track numbers are zero-based but sector numbers start from one.)

We can convert the type DISK into a linear array of SECTOR as follows:

```
type
     DISK = array [0..(TRACKS_PER_DISK*SECTORS_PERTRACK)-1] of
       SECTOR;
```

9

We use this linear representation for addressing the disk by physical sector number (PSN). The relations between the PSN and track and sector numbers are:

```
PSN = (TRACK_NUM * sectors_per_track) + SECTOR_NUM - 1;
TRACK_NUM = PSN div sectors_per_track;
SECTOR_NUM = (PSN mod sectors_per_track) + 1;
```

2.3.1.2 Physical Sector Size

Any physical sector size may be accommodated. An I/O request in physical sector made simply causes a full sector to be transferred. The Pascal programmer is responsible for ensuring that the data area is at least large enough for one physical sector.

Programs written using physical sector mode are not expected to be portable to different disk hardware without some modification.

This page last regenerated Sun Jul 25 13:37:38 2010.

3.0 The Interpreter Level: RSP/IO

This section provides details of the design and operation of the Input/Output division of the Runtime Support Package (RSP/IO). While the design itself is processor and hardware independent, it is intended to be realized in native code. Thus the final product will be processor-specific but still independent of the exact peripherals used. 10

3.1 Calling Mechanisms

Here are the details of how each routine in RSP/IO is called from the Pascal level. The level of detail is intended to be such that an implementor of RSP will know how to get parameters off the stack when RSP is called and how the stack should look when RSP returns. The detailed semantics of each routine are discussed in section 3.2.

3.1.1 UNITREAD and UNITWRITE

```
PROCEDURE UNITREAD(UNIT_NUMBER: INTEGER;
                   VAR DATA_AREA: PACKED ARRAY [0..BYTESTOTRANSFER-1]
                      OF 0..255;
                   BYTES_TO_TRANSFER: INTEGER;
                   [ LOGICAL_BLOCK: INTEGERS;
                   [ CONTROL: INTEGER ]]
                  );

PROCEDURE UNITWRITE(UNIT_NUMBER: INTEGER;
                    VAR DATA_AREA: PACKED ARRAY [0..BYTESTOTRANSFER-1]
                       OF 0..255;
                    BYTES_TO_TRANSFER: INTEGER;
                    [ LOGICAL_BLOCK: INTEGERS;
                    [ CONTROL: INTEGER ]]
                   );
```

3.1.1.1 Parameter Description

UNIT_NUMBER has been discussed in section 2.1.1. DATA_AREA is the user's buffer to or from which the data will be transferred. Describing it. as a VAR parameter signifies that UNITREAD and UNITWRITE are passed a pointer to the start of the data area. The Pascal programmer will call, say, UNITWRITE with an *array element*, for example `A[2]`, as the actual parameter. Thus the procedure is provided with the *starting address* for the transfer. For byte-oriented units, the address of the start of the data area may or may not be on a word (16 bit) boundary. In the case of block-structured (disk) units, however, it is only defined in the case that it *is* on a word boundary; that is, a Pascal programmer must not allow actual parameters which reference non word-aligned bytes to occur when transferring to/from the disk. This is to avoid restricting block-structured units to byte-by-byte transfers.

Starting with the 2.0 release level, byte addresses such as that described above are represented as an *address couple*, consisting of a *word base address* and a *byte offset*. On processors which use byte addressing, the effective address is computed by simply adding the *base* and *offset*, since both quantities are in bytes. For processors using word addressing, the effective address is computed by indexing byte-wise from the base address (always toward higher locations). 11

Note: For release Level I systems, the data area address is represented by a single word, *i.e.* by a simple byte address rather than an address couple.

The third item in the READ or WRITE parameter list, `BYTES_TO_TRANSFER`, contains the number of bytes to move between the user's data area and the physical unit.

Two optional parameters follow for UNITREAD and UNITWRITE: `LOGICAL_BLOCK` and `CONTROL`. If not specified by the Pascal programmer, the compiler will assign them both the default value zero. `LOGICAL_BLOCK` is only relevant for block structured units; as discussed in section 2.3, it specifies the Pascal logical block to be accessed. The `CONTROL` word has been discussed in section 2.1.2.

3.1.1.2 Parameter Stack Format

`UNITREAD` and `UNITWRITE` receive their parameters on the evaluation stack in the following order (each box represents a 16-bit quantity): 12

```
++++  |///////////////| <- - - - (when finished, SP
      +---------------+           points here)
      |  Unit Number  |
      +---------------+
      |  Word Base    |
      +---------------+
      |  Byte Offset  |
      +---------------+           (The stack shown here
      |  Byte Count   |               grows down)
      +---------------+
      | Block Number  |
      +---------------+
      |   Control     | <- - - - SP
----  +---------------+
```

Figure 3.0 — Stack state on entering UNITREAD or UNITWRITE

Like ordinary Pascal procedures, these RSP routines pop their parameters from the stack when they are finished.

3.1.2 UNITBUSY

```
FUNCTION UNITBUSY(UNIT_NUMBER: INTEGER): BOOLEAN;
```

On implementations supporting asynchronous I/O, this function tests whether the specified unit is busy or not and returns the boolean result as the function value. On the totally synchronous system that we are describing, `UNITBUSY` should always return *false*. Figure 3.1 illustrates the stack states before and after calling UNITBUSY; notice that the stack pointer does not change.

13

```
++++ |///////////////|                |///////////////|
     +---------------+                +---------------+
     |  Unit number  | <---- SP ----> |   false (0)   |
     +---------------+                +---------------+
          before                            after
```

Figure 3.1 — Stack state before and after `UNITBUSY`

3.1.3 UNITWAIT

```
PROCEDURE UNITWAIT(UNIT_NUMBER: INTEGER);
```

Like `UNITBUSY`, `UNITWAIT` is only useful in an asynchronous environment. It is intended to kill time until the designated unit becomes not busy. In a synchronous system, `UNITWAIT` is essentially a no-op since no unit should be busy unless a read or write request is pending. The single parameter is on top of stack when the procedure is called and is popped off before the procedure returns. The use of the stack is illustrated in Figure 3.2.

```
++++ |///////////////|       SP ----&gt |///////////////|
     +---------------+                 +---------------+
     |  Unit number  | <---- SP
     +---------------+
          before                             after
```

Figure 3.2 — Stack state before and after `UNITWAIT`

3.1.4 UNITCLEAR

```
PROCEDURE UNITCLEAR(UNIT_NUMBER: INTEGER; UINITPTR: ^UIR);
```

The purpose of `UNITCLEAR` is implied by its name: it restores the specified unit to its “initial” state. In an asynchronous system, this implies cancelling any pending I/O operations. In the synchronous environment with which we are concerned here, it is useful for initializing the RSP and BIOS routines concerned with that unit. The two parameters, `UNIT_NUMBER` and `UINITPTR` are, respectively, the number of the unit to be initialized and a pointer to a *Unit Initialization Record* (UIR) containing unit-specific initialization values. The structure of the UIR may vary depending on the type of unit.

If the value of `UINITPTR` is *nil* then RSP/IO must provide a pointer to a default UIR. The pointer is then passed to the BIOS. A Pascal representation of the UIR structure is shown in Figure 3.3. This corresponds to a 28-byte physical structure. The structures of the various cases are diagrammed In Figure 3.4. An area of six bytes has been reserved for future use. Note that RSP/IO *must* set UBREAKVECTOR (in the case UNITKIND = UCONSOLE) regardless of the contents of that field assigned by the Pascal system. In practice, the Pascal programmer will leave UBREAKVECTOR uninitialized, knowing that only the interpreter knows the address of the BREAK-handling subroutine. 14

Correct interpretation of this Pascal representation requires the knowledge that, in UCSD's implementation, the values used to represent the values of variables of scalar types such as unit_types are zero-based starting from the left. Thus a variable of type unit_types having the value uconsole actually has the value zero, one means uprinter, two means uremote and three means ublocked. The implementation is similar for all other scalar types.

```
type
  unit_types = (uconsole, uprinter, uremote, ublocked);
  baud_types = (b_11O, b_300, b_600, b_1200, b_2400,
                b_4800, b_9600, b_19200, b_autosense, b_other);
  parity_types = (p_even, p_odd, p_none);
  stp_bit_types = (s_one, s_oneandhalf, s_two);

  uir = record
```

```
      case UNITKIND: unit_types of
        uconsole,
        uprmnter,
        uremote:
          (UDATABITS: integer;
           USTOPBITS: stp_bit_types
           UBAUDRATE: baud_types;
           UPARITY: parity_types;
           USPECIAL: integer; (* Used with b_other *)
             (* Future use area *)
           UFUTURE: array [0..2] of integer;
```

15

```
      case UNITKIND of
        uconsole:
          (USTARTSTOP: char;
           UFLUSH: char;
           UBREAK: char;
           UALPHALOCK: char;
           UBREAKVECTOR: ^integer);
        uprinter:
          (USTARTSTOP: char;
           UFLUSH: char;
           UPAGELINES: integer)
      );
   (* ublocked needs none *)
end;
```

Figure 3.3 — Sample UIR Declaration

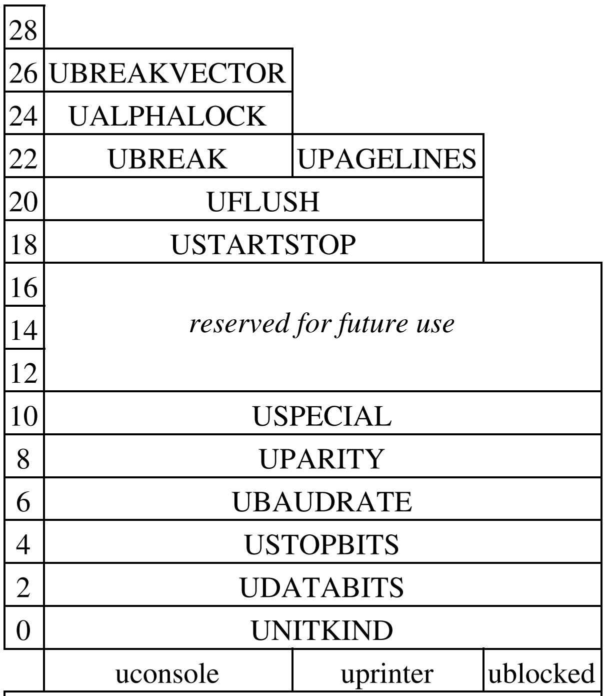

Figure 3.4 — UIR Physical Structure

When RSP/IO is passed a *nil* UINITPTR, it should provide the BIOS with default UIR's having the values shown in Table 3.0.

console	UNITKIND = 0	(* uconsole *)
	UDATABITS = 8	(* eight *)
	USTOPBITS = 1	(* one and a half *)
	UBAUDRATE = 6	(* b_9600 *)
	UPARITY = 2	(* p_none *)
	USPECIAL = 0	(* not needed *)
	USTARTSTOP = 19	(* DC3 *)
	UFLUSH 6 (* ACK *)	
	UBREAK = 0 (* NUL *)	
	UALPHALOCK 18 (* 0C2 *)	
	UBREAKVECTOR = Address of Break Subroutine	
printer	UNITKIND = 1	(* uprinter *)
	UDATABITS = 8	
	USTOPBITS = 1	
	UBAUDRATE = 1	(* b_300 *)
	UPARITY = 2	
	USPECIAL = 0	(* not needed *)
	USTARTSTOP 19	(* DC3 *)
	UFLUSH = 6	(* ACK *)
	UPAGELINES = 58	(* 11 in., 6 lpi, 4-line margins *)
remote	UNITKIND = 2	(* uremote *)
	UDATABITS = 8	
	USTOPBITS = 1	
	UBAUDRATE = 6	
	UPARITY = 2	
	USPECIAL = 0	(* not needed *)
disk	UNITKIND = 3	(* ublocked *)

Table 3.0 — Default UIR Values

3.2 Semantics

This section will detail the processing to be performed by RSP/IO. The primary function of RSP/IO is to manage calls to BIOS. In the case of disk I/O, for example, RSP does little except call BIOS to do all the work. Secondarily, RSP/IO is responsible for handling certain special functions which shall be described here. Appendix A contains a Pascal realization of RSP/IO which should be considered the most precise reference for the semantics.

3.2.1 Special Character Handling on Output

Output to the printer, console or remote units must be massaged to properly handle Blank Compression Codes and CR's. 17

3.2.1.1 Blank Compression Code (DLE's)

The UCSD Pascal System supports text files containing a two-byte blank compression code. It is the responsibility of RSP/IO to decode the blank compression code and send an appropriate number of blanks. The first byte is an ASCII DLE (decimal 16) which signals that the next byte should be interpreted as being (the number of blanks to be sent)+32. Thus the next byte following the OLE should be processed by subtracting 32 from its value and sending that number of blanks.

3.2.1.2 Carriage Return - Line Feed

Text files contain ASCII CR's (decimal 13) at the end of lines. We define this character as meaning "New Line", *i.e.* a carriage return followed by a line feed. Thus it is the responsibility of RSP/IO to send an ASCII LF (decimal 10) after sending each CR.

3.2.1.3 NOSPEC Bit in CONTROL Parameter

When this bit is set, the special handling accorded DLE's and CR's is shut off and they are sent out like other characters.

3.2.2 Special Character Handling on Input

There are several characters which will receive special treatment coming from the console in a complete implementation of this I/O system. All but one of them, however, are handled by the BIOS. The one which is handled in RSP/IO is the unit EOF character.

3.2.2.1 Unit EOF Character

The console EOF character, when received from the keyboard, printer or remote ports, signals that "end-of-file" has bean reached on that particular unit. Rather than being a fixed ASCII code, this is a 18 "soft character". That is, the exact character code which will be interpreted as "Console End-Of-File" may be changed during system execution by the Pascal user. Further discussion of the soft characters used by the I/O Subsystem may be found in section 4.4. The EOF character is in the SYSCOM data area and must be accessed by RSP/IO to determine what character to look for. When the EOF character is found in the input stream, the action to be taken depends somewhat upon which unit was referenced. If we are reading from unit 1 (CONSOLE), then a NUL (character code 0) is returned to the user's buffer instead of the EOF character. For all other units, the EOF character is put in the user's buffer. In either case, no further characters are transferred to the buffer; control immediately returns to the Pascal level. Further details are in Appendix A (procedure `READBYTES`).

3.2.2.2 BIOS Functions

Of the remaining special input characters, START/STOP, FLUSH, ALPHALOCK and BREAK, two (ALPHALOCK and BREAK) are used only for input from the console, not from the printer or remote ports. The other two (START/STOP and FLUSH) may be handled from both console and printer, but not from remote. They are handled by the BIOS and are described in section 4.5.1.4.

3.2.2.3 NOSPEC Bit in CONTROL Parameter

As in 3.2.1.3 above, when this bit is on, the special character handling performed by RSP/IO is turned off. This includes the EOF sensing function described above. It does not affect the BIOS functions.

3.3 Modelling RSP/IO Using Pascal

As the reader will notice in Appendix A, a Pascal program has been written which performs all special character handling required and calls BIOS with the specified parameters. While no version of the UCSD Pascal System has been implemented using this Pascal code for RSP/IO, it could be done.

Our real intention in providing this program is to provide a precise specification of the RSP/IO requirements to those who must implement it in native code. It is possible to translate the Pascal into assembly language and produce an implementation that is quite efficient if the implementor is not too literal-minded. 19

This page last regenerated Sun Jul 25 13:37:38 2010.

4.0 The Machine Level: BIOS

As explained in section 1.1, the Basic Input/Output Subsystem is responsible for providing the actual access to I/O devices. Both the design and implementation of the BIOS is specific to a given processor and I/O configuration. In this section we will attempt to specify the nature of BIOS in sufficient detail for an experienced programmer, in cooperation with IIS personnel, to write the code for a given processor and set of peripherals. 20

The general scheme discussed below uses vectors from RSP/IO to BIOS subroutines for reading, writing and initializing. The exact vector scheme and means of passing parameters must be worked out separately for each processor. Arrangements that have already been worked out for certain processors are given in Appendix C.

4.1 Design Goals

The speed of the BIOS code is generally of small significance compared to the speed of the I/O devices which it serves. When peripherals are changed, which may occur frequently, it will often prove that only minor changes need be made in an existing BIOS to service the new hardware. Also, since BIOS is core-resident, each byte it occupies is one less available to the Pascal user. For these reasons, we suggest that major design goals be (1) compactness and (2) clarity.

Like the rest of the Interpreter, the BIOS should be ROM-able. Obviously it will also require access to some RAM. It should be possible to easily change the addresses involved using some equates and thus reassemble the BIOS for a given memory configuration.

4.2 Completion Codes

All read, write or initialization calls to BIOS must return a byte to RSP containing status information on the I/O request just serviced. The value of this byte is the "completion code" discussed in section 2.2. Most of the standard completion codes listed in 2.2 are not relevant to BIOS — they are returned by the Pascal Operating System for file errors and the like. The following errors can be returned by BIOS: 21

Code	Meaning
0	No error
1	CRC error
3	Illegal operation on unit
9	Unit not on line
16	Write attempt to write-protected disk
17	Illegal Block or Sector number
18	Non-zero byte count (physical sector mode)
19	Error in UIR
100..199	*Hardware dependent*

All other errors are considered hardware-dependent. For these, BIOS should return codes in the range 100..199. The selection of appropriate codes is left to the BIOS writer.

Note that any units not implemented **must** arrange to return a completion code of 9 ("Unit not on line") when an attempt is made to initialize or use them.

4.3 Calling Mechanisms

In this section we discuss the parameters required in the BIOS calls for each unit. Each unit has three BIOS calls associated with it: READ, WRITE and INIT. Each unit has varying needs for information associated with these functions. Remember that **all** calls must return the completion code byte. For a summary of the BIOS calling requirements, see Appendix B.

4.3.1 Console

Only one parameter is needed For reading and writing, containing the data byte to be transferred. Initialization of the console BIOS requires that the identity of a number of special control characters be provided in the UIR, as well as serial line interface settings and a BREAK vector. The details of handling these special characters are discussed in section 4.5.1.5.

4.3.2 Printer

A single parameter is passed: a byte containing the data read or written. Initializing the printer requires 22
the serial line interface settings in the UIR. Optional use may be made of the UIR field UPAGELINES which specifies the number of lines to be printed per page. If `UPAGELINES = 0` then no page breaks should be made by BIOS.

4.3.3 Disk

The calling mechanism for disk units requires five parameters for reading and writing:

1. a starting logical block number as described above,
2. a count of the number of bytes to transfer (unsigned 16 bits, *i.e.* 0 to 64K),
3. the address of the data area to transfer to or from,
4. a drive number (0..n-1, given *n* drives; currently `n = 6` is assumed), and
5. the CONTROL word.

It should be noted that, in the case of disk I/O, the data area address is guaranteed to be on a word boundary (an even byte address) and the number of bytes to transfer will always be even. On initialization, the UIR is empty.

4.3.4 Remote

The remote unit requires a single parameter for reading and writing: a byte containing the data being transferred. When initializing the remote unite the UIR contains serial line interface settings.

4.4 Character Codes

The U.C.S.D. Pascal system assumes that the printer and console units will support the use of ASCII printable characters and a few standard control codes (CR, LF, SP, NUL and BEL). The remaining control codes which may be useful (*e.g.* cursor positioning and screen erasure) are “soft” characters which may be changed by the Pascal user (by running the SETUP utility) to suit the requirements of his current hardware. The reason for inflicting these hardware dependencies upon the Pascal level is the simple fact of life that terminals use control codes which vary widely and we want to be able to change terminals without installing a new BIOS. The basic issue is one of mapping logical control symbols into the control codes recognized by the hardware.

Suppose, for example, that there is pre-declared procedure CURSORBACK which causes the cursor on a screen terminal to move left one column. Somewhere in the system, CURSORBACK must cause a control code to be sent to the terminal which will cause the desired responses whether it's control-U, control-H or an escape sequence. One wag to do this would be for the Pascal level to emit a standard code which the BIOS then translates into whatever is correct for the current terminal. This has the disadvantage of requiring a new BIOS for every slightly different terminal. The approach which we have taken sees to it that the correct code is sent to BIOS for the current terminal on line. The details of how this is done are irrelevant to the I/O Subsystem and are elsewhere in the UCSD Pascal System documentation. 23

Due to the capability of many devices to make use of eight-bit control codes, the Pascal system makes no assumptions as to the relevance of the high-order bit and transfers the whole byte faithfully. When using [seven-bit] ASCII, the value of the high-order bit is defined to be zero. In other words, the code for the character ‘A’ must be 65 (decimal) rather than 193 (or 41 hex rather than C1, if you prefer). This has the effect of requiring BIOS to return ASCII codes with the high-order bit off for all the standard characters.

RSP will be sending both upper and lower case characters to BIOS. Thus for upper-case-only display devices that do not display lower case codes as upper case. BIOS must map lower case into upper case.

4.5 Semantics

24

4.5.1 Console

Here we discuss the required and optional features of the console device. The console device is assumed to be a CRT terminal in the following discussion, although a typewriter device may also be used.

4.5.1.1 Output Requirements

As noted in section 4.4, we *depend* on the action of certain ASCII control codes. These are the minimum requirements for a console device:

CR <carriage return> (hex OD)
: Move cursor to the beginning of the current line (column 0).

<LE line feed> (hex OA)
: Move cursor to the next line down while the column position remains the same. Starting from any but the last line on the screen, the contents of the screen should remain the same while the cursor moves downward. If the cursor is on the last line when the LF is issued, it should remain in the same position while the rest of the display scrolls upward one line and the bottom line clears.

DEL <bell> (hex 07)
: If an audio signal is available, it should be sounded. If one is not available, the terminal should do nothing. The delay time required while doing nothing is not significant.

SP <space> (hex 20)
: Write a space at the current cursor position (erasing whatever is there) and advance the cursor position by one column. If the cursor is already at the last position in a line, the position of the cursor after the SP is undefined. We prefer that the cursor remain in its prior position in this case. If the cursor is in the last column of the last line on the screen, not only is the position of the cursor undefined after the SP, but so is the state of the screen: maybe it scrolled and maybe it didn't. As above, we would prefer

that the cursor remain where it was and that the screen not scroll.

NUL <null> (00) 25

Delay for the time required to write one character. The state of the console should not change.

Any printable character

Same as the discussion for SP, except, of course, write the character and not SP!

Note that the effect of sending non-printable characters other than those described above to the screen BIOS is not defined since it is known to vary from terminal to terminal.

4.5.1.2 Output Options

The following set of cursor and screen functions should be provided if possible, however they are optional in the sense that almost all major functions of the UCSD Pascal System will still be available if they are not provided. The control characters or sequences of characters which provide these functions are left unspecified for the reasons described in section 4.4.

Reverse Line Feed:

Move the cursor to the next line higher on the screen without changing column or the other contents of the screen. If the cursor is already on the top lines the result is undefined. If possible, the screen should reverse-scroll in such a case, or if that is not feasible, the cursor and screen should just remain as they were.

Non-destructive Forward and Backward Space:

Move the cursor in the direction indicated without changing the contents of the screen (*i.e.* move it non-destructively). The position of the cursor is undefined if an attempt is made to move it beyond the end of a line. The preferred result is that cursor and screen remain unchanged in such a case.

Cursor Home:

Move the cursor to the upper left-hand corner of the screen without changing the other contents of the screen.

Cursor X,Y Positioning:

Move the cursor to some absolutely determined row and column without disturbing the contents of the screen. The result is undefined if an attempt is made to move the cursor to a non-existent position.

Erase to End of Screen: 26

Erase from the cursor position to the end of the screen, leaving the cursor where it started and the other contents of the screen undisturbed.

Erase to End of Line:

Erase from the cursor position to the end of the current line, leaving the cursor where it started and the rest of the screen undisturbed.

4.5.1.3 Input Requirements

Input from the keyboard should **not** be echoed to the screen by BIOS; this function will be handled by RSP/IO. Keys which represent ASCII characters should generate eight bit codes between 0 and 127. Problems were encountered with an implementation in which an early form of BIOS returned ASCII with the high bit set (*i.e.* between 128 and 255). In that instance, the high bit had to be turned off by RSP/IO before the character was used. Other [non-ASCII, *e.g.* special function] kegs can generate codes between 128 and 255 if desired.

4.5.1.4 Input Options

If possible, we recommend that the console input BIOS be responsible for the following special functions:

4.5.1.4.1 START/STOP

The START/STOP character is used to control console output. When START/STOP (a soft character) is received, console output is suspended until (a) another START/STOP character is received or (b) the BREAK character is received. Action to take in the latter case is discussed below. Should the former case occur, the suspended activities should resume exactly as they left off. The chief benefit gained through this arrangement is to enable the user to suspend console output processes which are proceeding faster than he would like, *e.g.* a text file scrolling across the screen at 9600 baud. The suspension process takes place wholly within BIOS and requires no communication to RSP. Note that the queueing of keyboard input should continue during the suspension.

4.5.1.4.2 FLUSH

FLUSH is another soft control character; when FLUSH is typed, the console output BIOS throws away 27
all output characters (*i.e.* does not display them now or ever) until FLUSH is typed again, input is requested from the console BIOS or the console BIOS is reinitialized. This feature is useful when a long text file is being displayed on the console and you're tired of looking at it. Push FLUSH and it terminates rather quickly. It is also useful when a process is generating console output which significantly slows the rate at which the process proceeds.

4.5.1.4.3 ALPHALOCK

Keyboards supporting both upper and lower case characters should have an alpha-lock facility; something which causes all alphabetic keys to generate upper case without shifting the other keys. If the hardware does not support such a feature (*i.e.* an alphalock key), it should be done by BIOS. It should be implemented as a “toggle” controlled by the ALPHALOCK soft character.

4.5.1.4.4 BREAK

The remaining special character to watch for is BREAK, also a soft character. When BREAK is typed, the console input BIOS should immediately give control to a special RSP routine. The vector to this special routine will be passed at console initialization time. Note that receipt of BREAK should terminate any START/STOP suspension pending.

4.5.1.4.5 Type-Ahead

When non-special (*i.e.* not described in the section above) characters are received from the keyboard with no read request pending, they should be queued until the next read request, which should be serviced from the queue. When characters in excess of the maximum queue size are received, they should be ignored and the queue remain intact. While a type-ahead of even one character is better than none at all, we recommend a minimum queue capacity of about 20 characters, up to a maximum of about 80. If possible the bell should be sounded for each character entered from the keyboard after no room remains in the queue.

When operating with a keyboard that is not interrupt-driven, it is possible to provide type-ahead by 28
polling the console status at strategic locations elsewhere in the BIOS. This will work fairly well if the current process is spending a lot of time doing I/O, however characters may well get lost. For this reason we suggest that only the BREAK character be sensed in this manner. Complete type-ahead may be done this way at the user's own risk.

4.5.1.5 Initialization

Initialization of the console BIOS will make use of the information in the UIR for several purposes:

(1) *Serial Line Settings*
    If the serial line interface hardware provides software-selection of any of its parameters, they should be set in accordance with the UIR.
(2) *Soft Control Character Recognition*
    The BIOS should use the character codes in the UIR to set the control characters it is sensitive to.
(3) *BREAK vector*
    The UIR contains the address of the Interpreter subroutine which will recover from user BREAK requests.

The structure of the console UIR is shown in figure 3.4. Initialization should also cause any START/STOP and FLUSH flags to be cleared and any characters currently in the type-ahead queue to be discarded.

Note that the console display should remain unchanged after console initialization. Specifically, the BIOS should NOT issue a clearscrean during initialization. The Pascal system is responsible for issuing clearscreens when needed.

4.5.2 Printer

The Printer unit is conceived of as being a line printer or other hard copy device. Any ASCII display device may be used in actuality.

4.5.2.1 Output Requirements

29

In order to serve the widest variety of hard copy devices, RSP/IO does not buffer a line of text and send it all at once. Rather it sends the printer BIOS a single character at a time. Since some line printers must buffer a line and then print it all at once, this becomes a BIOS requirement if such a device is to be served. Thus, in order to determine when a line is finished, the BIOS must recognize certain line delimiter characters. These characters may have additional meaning besides just being line delimiters. They are summarized:

Line Delimiters

CR <carriage return> (hex 0D)
    Print the line. An automatic line feed should NOT be done. If the hardware requires that a line feed be performed, then if the next character is a line feed it must be ignored.
LF <line feed> (hex 0A)
    In normal operations RSP/IO will only send LF's to BIOS immediately after a CR. Should a LF be sent which is NOT preceded by a CR, it must be interpreted as a line delimiter. If the hardware allows a simple line feed to be performed (without a return) then this should be done. If, as would be the case with a line printers a complete “new line” operation (return and line feed) is all that can be done, then this may be allowed.
FF <form feed> (hex 0C)
    The printer should advance the paper to top-of-form if possible and perform a carriage return. If no such feature is available, the printer may execute a “new line” operation, *i.e.* a return followed by a line feed.

4.5.2.2 Input Requirements

The printer is allowed to talk back to the Pascal system in much the same way as the console or remote units. The printer BIOS may watch for START/STOP or FLUSH control characters coming from the printer if desired.

4.5.2.3 Initialization

Initialization of the printer should make sure that it is ready to print at the beginning of a blank line, thus 30
a “new line” (return and line feed) operation *may* be in order. Any characters which have been buffered but not printed are lost. It is not desired that the printer perform a form-feed each time it is initialized.

4.5.3 Disk

4.5.3.1 Mapping Logical Blocks onto Physical Sectors

4.5.3.1.1 Interleaving

A primary function of the disk BIOS is mapping the 512 byte Pascal logical blocks onto one or more physical sectors of some arbitrary size (see section 2.3). In the simplest possible schemes the disk has 512 byte sectors and logical block numbers are identical to physical sector numbers. The most common situation (IBM format floppy disks) finds us with 128 byte physical sectors. Here, the simplest mapping would establish the correspondence between a logical block and four consecutive sectors. Due to limitations in disk hardware, however, it is quite common to *interleave* “logical sectors” on the disk.

When Interleaving is used, the disk driver, when asked for, say, eight consecutive sectors, may actually transfer sectors 1, 3, 5, ..., 15. Since the same interleaving algorithm is used for both reading and writing, the interleaving is transparent to the user (until he tries to use his disk on a system using different interleaving). The advantage in using interleaving is that the hardware may not be fast enough to pick up physically contiguous sectors on a single disk revolution, but, with an appropriate interleaving algorithm, will be able to pick up sectors that are *logically* contiguous, though not physically contiguous, on a single revolution.

The UCSD Pascal system makes no assumptions about the interleaving method used by the BIOS, except that it works.

4.5.3.1.2 Bootstrap Location

While bootstrap schemes vary, typical UCSD Pascal implementations make use of a hardware (usually 31
ROM) bootstrap to load and execute a primary software bootstrap which, in turn, loads and executes a secondary software bootstrap. The secondary bootstrap then loads the Pascal interpreter and operating system, performs required initialization and starts UCSD Pascal. To be accessible to the hardware bootstrap, the primary software bootstrap must reside at a location on the disk which is predetermined by the hardware vendor. Since these locations can vary widely, it is necessary for UCSD's physical disk format requirements to be flexible in this regard.

There are two primary requirements which must be met: (1) The primary bootstrap area must not overlap disk data structures maintained by the Pascal system and (2) The primary bootstrap area must be accessible to the Pascal system to facilitate maintenance of the bootstrap code.

The Pascal system reserves logical blocks 0 and 1 for bootstrap code, thus allowing 1024 bytes in the interleaving format used on the rest of the disk. Thus the simplest solution is to map the Pascal logical blocks onto the disk so that the primary and secondary bootstraps are together in blocks 0 and 1.

If 1024 bytes is not enough, or if the interleaving format is unacceptable to the hardware bootstrap, then the primary bootstrap area must be outside of the “Pascal disk”. The Pascal logical blocks must be mapped onto the disk in such a way that the hardware-defined bootstrap area is *inaccessible* to the Pascal system *as a logical block* (it will still be accessible in physical sector mode).

The details of the bootstrap procedure are not discussed in this document.

4.5.3.2 Output Requirements

Nothing fancy here, simply transfer however many physical sectors are needed to accommodate the data. To make it simple, after a disk-write in which `(BYTESTOTRANSFER mod 512)` is not equal to zero (*i.e.* the last block is partially written to), the remaining contents of the last block are undefined. This makes it possible to write whatever garbage remains in a buffer if that is convenient to fill up a whole sector. Figure 4.0 illustrates this situation. The Pascal level is responsible for keeping track (in logical block numbers and byte counts) of where the good data is.

Example: Write to disk.

Number of bytes to transfer =	1174
Starting logical block number =	72
Data area address =	(irrelevant)

```
+-------------------+-------------------+-----:-------------+
|                   |                   |     :             |
|     Block 72      |     Block 73      |     Block 74      |
|    (512 bytes)    |    (512 bytes)    | 150 : 362 bytes   |
|<==================data==================>: (undefined) |
|                   |                   |     :             |
+-------------------+-------------------+-----:-------------+
:start of data                     end of data:             :
                                           end of last block:
```

Figure 4.0 — State of Blocks on Disk After Being Written To

4.5.3.3 Input Requirements

On input from the disk, it is certainly not permissible to overwrite the end of the assigned data area! Therefore, BIOS is responsible for transferring exactly the number of bytes requested. This can probably best be accomplished by buffering the last sector and then transferring that part of it which was requested.

4.5.3.4 Initialization

Initializing a disk unit should bring it to a state in which it is ready to read or write from/to any given track or sector. For some drives with simple controllers, the head should be stepped to track 0 to facilitate the BIOS disk driver's remembering the current track.

4.5.3.5 Physical Sector Mode

When the PHYSSECT bit of the CONTROL word is set, disk access should be performed in the manner described in section 2.3.1. The BIOS is responsible for returning a completion code of 18 if the byte count is not zero. 33

4.5.4 Remote

This unit is intended to be an RS-232 serial line for supporting various types of communication.

4.5.4.1 Output Requirements

Output is made a single byte at a time.

4.5.4.2 Input Requirements

If interrupt-driven, input should be captured in a “type-ahead” buffer similar to that used for the console unit. The buffer should be 80 or more bytes long.

4.5.4.3 Initialization

On initialization, the remote BIOS is passed a pointer to a UIR. The structure of the UIR for the remote unit is shown in figure 3.4. If software selection of any of these serial interface settings is possible then it should be done.

4.6 Special BIOS Calls

These functions are provided by the BIOS to make configuration-specific functions accessible to the Interpreter. Although these functions are not related to Input/Output, they are put in the BIOS as the repository for configuration-specific code.

4.6.1 Memory Sizing

Since the Interpreter is designed to run in various sizes of memory, it must know the address of the last accessible word. A call to the memory sizing function of BIOS should return this address. Note that a “word” address should be returned, *i.e.* on an 8080 system with 64k bytes of RAM the last byte address is ‘FFFF’, but the last word address is ‘FFFE’.

In many BIOS implementations, it will be possible to return a simple assembly-time constant. If, however, dynamic memory sizing must be performed, the sampling routine should restore the contents of memory sampled. 34

4.6.2 System Halt

The system halt routine in BIOS should perform whatever is necessary to terminate Pascal execution in an orderly fashion, e.g. eject disks. Note that the Pascal system itself will already have taken care of Pascal-ish details such as closing files.

4.6.3 Start Clock

If the system is equipped with a real-time clock it should be started, otherwise this call may be ignored.

This page last regenerated Sun Jul 25 13:37:38 2010.

Appendix A

Note: Appendix A has been temporarily deleted.

35

This page last regenerated Sun Jul 25 04:12:13 2010.

Appendix B

36

Summary of BIOS Calling Sequences

Following in Figure B.0 is a summary of what has been described in section 4.3. Processor-specific protocols for certain machines are provided in Appendix C. All calls to BIOS return a completion code.

Entry Point	Parameters
CONSOLEREAD	single data byte
CONSOLEWRITE	single data byte
CONSOLEINIT	UIR pointer
PRINTERREAD	single data byte
PRINTERWRITE	single data byte
PRINTERINIT	UIR pointer
DISKREAD	block no. byte count data area address drive no. CONTROL word
DISKWRITE	block no. byte count data area address drive no. CONTROL word
DISKINIT	drive no. UIR pointer
REMOTEREAD	single data byte
REMOTEWRITE	single data byte
REMOTEINIT	UIR pointer

37

MEMSIZE	Address of last RAM word
SYSHALT	*(none)*
CLOCKSTART *(none)*	

This page last regenerated Sun Jul 25 04:12:13 2010.

Appendix C

Examples of Current Processor-Specific BIOS Calling Sequences 38

C.1 8080/Z-80

Entry Points:

All BIOS entry points are given as offsets from the beginning of the BIOS code space. These locations should contain a JMP instruction to the appropriate address in BIOS.

Parameters:

When parameters are not being passed in a specified register, they are pushed on the stack. Offsets from top-of-stack are given, recognizing that the stack grows down.

Completion Code:

Return in register C.

Calling Sequence:

RSP will use the CALL instruction to call BIOS. Thus the return address is at (SP),(SP)+1. All registers are available for use by BIOS. BIOS should clean off the stack before returning to RSP.

Entry Point	Offset (hex)	Parameters
CONSOLEREAD	00	return data byte in Reg C
CONSOLEWRITE	03	write data byte in Reg C
CONSOLEINIT	06	UIR pointer at (SP)+2,(SP)+3
PRINTERREAD	09	return data byte in Reg C
PRINTERWRITE	0C	write data byte in Reg C
PRINTERINIT	0F	UIR pointer at (SP)+2,(SP)+3
DISKREAD	12	block no. at (SP)+2,(SP)+3 byte count at (SP)+4,(SP)+5 data area addr. at (SP)+6,(SP)+7 drive no. at (SP)+8 CONTROL byte at (SP)+9
DISKWRITE	15	block no. at (SP)+2,(SP)+3 byte count at (SP)+4,(SP)+5 data area addr. at (SP)+6,(SP)+7 drive no. at (SP)+8 CONTROL byte at (SP)+9
DISKINIT	18	drive no. in Reg C UIR pointer at (SP)+2,(SP)+3
REMOTEREAD	1B	return data byte in Reg C
REMOTEWRITE	1E	write data byte in Reg C
REMOTEINIT	21	UIR pointer at (SP)+2,(SP)+3

39

C.2 6500 Series

Entry Points:
All BIOS entry points are given as offsets from the beginning of the BIOS code space. These locations should contain a JMP instruction to the appropriate address in BIOS.

Parameters:
When parameters are not being passed in a specified register, they are pushed on the stack. Offsets from the address pointed to by S (described as (S)) are given, recognizing that the stack grows down and that S normally points to the first available address below valid data.

Completion Code:
Return in register X.

Calling Sequence:
RSP will use the JSR instruction to call BIOS. Thus the return address is at (S)+1,(S)+2. All registers are available for use. The stack should be cleaned off by BIDS before returning to RSP.

Entry Point	**Offset (hex)**	**Parameters**
CONSOLEREAD	00	return data byte in Reg A
CONSOLEWRITE	03	write data byte in Reg A
CONSOLEINIT	06	UIR pointer at (S)+3,(S)+4
PRINTERREAD	09	return data byte in Reg A
PRINTERWRITE	0C	write data byte in Reg A
PRINTERINIT	0F	UIR pointer at (S)+3,(S)+4
DISKREAD	12	block no. at (S)+3,(S)+4
		byte count at (S)+5,(S)+6
		data area addr. at (S)+7,(S)+8
		drive no. at (S)+9,(S)+A
		CONTROL word at (S)+B,(S)+C
DISKWRITE	15	block no. at (S)+3,(S)+4
		byte count at (S)+5,(S)+6
		data area addr. at (S)+7,(S)+8
		drive no. at (S)+9,(S)+A
		CONTROL word at (S)+B,(S)+C
DISKINIT	18	drive no. in Reg A
		UIR pointer at (S)+3,(S)+4
REMOTEREAD	1B	return data byte in Reg A.
REMOTEWRITE	1E	write data byte in Reg A.
REMOTEINIT	21	UIR pointer at (S)+3,(S)+4

C.3 6800

Entry Points:
All BIOS entry points are given as offsets from the beginning of the BIOS code space. These locations should contain a JMP (extended) instruction to the appropriate address in BIOS.

Parameters:

When parameters are not being passed in a specified register, they are pushed on the stack. Offsets from the address pointed to by SP (described as (SP)) are given, recognizing that the stack grows down and that SP normally points to the first available address below valid data.

Completion Code:

Return in register B.

Calling Sequence:

RSP will use the JSR instruction to call BIOS. Thus the return address will be at (SP)+1,(SP)+2. All registers are available for use. The stack should be cleaned off by BIOS before returning to RSP.

Entry Point	**Offset (hex)**	**Parameters**
CONSOLEREAD	00	return data byte in Reg A
CONSOLEWRITE	03	write data byte in Reg A
CONSOLEINIT	06	UIR pointer at (SP)+3,(SP)+4
PRINTERREAD	09	return data byte in Reg A
PRINTERWRITE	0C	write data byte in Reg A
PRINTERINIT	0F	UIR pointer at (SP)+3,(SP)+4
DISKREAD	12	block no. at (SP)+3,(SP)+4 byte count at (SP)+5,(SP)+6 data area addr. at (SP)+7,(SP)+9 drive no. at (SP)+9,(SP)+A CONTROL word at (SP)+B,(SP)+C
DISKWRITE	13	block no. at (SP)+3,(SP)+4 byte count at (SP)+5,(SP)+6 data area addr. at (SP)+7,(SP)+9 drive no. at (SP)+9,(SP)+A CONTROL word at (SP)+B,(SP)+C
DISKINIT	18	drive no. in Reg A UIR pointer at (SP)+3,(SP)+4
REMOTEREAD	1B	return data byte in Reg A
REMOTEWRITE	1E	write data byte in Reg A
REMOTEINIT	21	UIR pointer at (SP)+3,(SP)+4

This page last regenerated Sun Jul 25 04:12:13 2010.