A C O R N S C I E N T I F I C

TECHNICAL REFERENCE

MANUAL FOR PANOS

PROVISIONAL ISSUE (EXTRACTS)

Inter-language Calling Standard
Object File Format
Calling Panos Standard Procedures from C
Pandora Definition Issue B

Issue B
11 June 1986
Acorn Computers Limited, Scientific Division

Panos Technical

User Document

10-Sep-85

Inter-Language Calling Standard

Contents

Change record

26-Apr-84	:	First Roff version.
21-May-84	:	Amendments made according to MT's comments.
09-Oct-84	:	Revised by MT (changes to string result mechanism).
22-Oct-84	:	Becomes Issue B.
24-Jul-85	:	Slight mods by MT, becomes Issue C
10-Sep-85	:	Further slight mods by MT, becomes issue D

Introduction

A number of compiled languages have been or are being implemented on Acorn products based on the National Semiconductor 32000-series microprocessors. These include Algol 68, BCPL, C, FORTRAN-77, IMP, Modula-2 and Pascal. It is obviously desirable for there to be as much compatibility between these languages as possible, so that the programmer is not restricted to writing all of his program in a single language. An example of the usefulness of this might be where a large statistical analysis program, written in Pascal because of the programmer's preference, might use a set of pre-written FORTRAN functions to do the maths processing, and some C library code to produce graphical output. The inter-language calling standard defined in this document is intended to facilitate this sort of flexibility.

The first part of this standard is a language-independent mechanism for the specification of the types of objects available, both as external data items, and as parameters and/or results of external procedures. It is intended that all conventional data and procedure types handled by languages conforming to the standard may be represented using this mechanism. The mechanism is implemented:

(a) by the compilers which generate the object modules, in providing this type information for each external item defined or referenced;
(b) by the system software controlling the inter-module linkage process, in ensuring type consistency between the modules being linked together.

This first part is defined as an appendix to the document:
"Acorn 32000-series Software: Acorn Object Format"

The second part - the actual cross-calling mechanism - is a definition, of the data structures and code sequences which must be generated by compilers so that external procedures may be called in a compatible manner. The present document describes this mechanism.

Note:

All inter-language calls under this standard will be mediated by a linker or other system facility, using Acorn 32000 Object Format (AOF).

Parameters

In the following discussion, the term "argument" refers to the data item actually pushed on the stack when a procedure is about to be called.

All arguments are passed on the SP stack (typically using the 32000's TOS addressing mode). Arguments are pushed consecutively, as either 4-byte or 8-byte items, before the procedure is called. The order of pushing is right-to-left, i.e. the last argument in the list is pushed first. Thus when the complete argument list is viewed as a sequence of store locations, the first argument is at the lowest address. Note that this is a definition of the order in which arguments are pushed, and not of that in which the parameters themselves are evaluated - the latter may or may not be by the particular language in which the call is written.

There are two basic categories of parameter; value and reference. In the former case, the argument is either the actual parameter value (a "direct parameter "), or a pointer to a store location containing it. In the latter, the argument is always a pointer to the parameter proper. Arguments are always either 4 or 8 bytes long. 4-byte arguments are used for passing pointers (addresses), integers of up to 32 bits, and single precision (-F) floating point numbers. 8-byte arguments are used to pass integers which require 33 to 64 bits, and double precision (-L) floating point numbers. "Direct parameters" are essentially intended for simple scalar items; structured items (e.g. arrays, strings, records) must be passed by pointer, as must any non-scalar parameter or one of more than 64 bits.

If a direct integer parameter occupies less than the full number of bits allocated for it (e.g. a byte, passed as a 4-byte argument), it must be passed at the least significant end of the argument. If such a parameter is to be treated as a signed integer, the top bits of the argument must sign-extended with a copy of the most significant (sign) bit of the parameter; otherwise they must be cleared to 0. In this connection, a pointer must be treated as a 32-bit unsigned quantity, and a boolean as a 1-bit unsigned quantity.

The treatment of specific types will now be dealt with.

Integers

When passed as a value parameter, an integer occupies 32 bits (standard doubleword) or 64 bits (quadword). A 64-bit integer is comprised of two contiguous doublewords, stored so that the less significant doubleword is at the lower address. When passing a 64-bit integer value parameter using MOVD instructions, the more significant doubleword must be pushed first to comply with this ordering. When passed by reference, the argument is a pointer to a store location laid out in the manner described above, i.e. less significant doubleword at lower address.

Floating point numbers

When passed by value, a 32-bit floating item is pushed onto the stack by the instruction:

```
MOVF item, TOS
```

so that 32 bits are pushed and the SP is reduced by 4.

When a 64-bit floating item is passed, the MOVL instruction is used rather than MOVF. Thus the item pushed is 64 bits long, and the SP is reduced by 8.

For a floating point parameter passed by reference, a 4-byte pointer is pushed. This is the address of a standard machine-format floating item of 32 (-F) or 64 (-L) bits.

Strings

String parameters under this standard may only be passed by value. Hence a called procedure must never modify a string parameter; however it is free to copy the string into its local data space and modify the copy, if the language in which it is written requires this (e.g. Algol 68 does not, but IMP does). This restriction may seen severe, but it arises because of the wide variety of ways in which strings are defined languages, and the consequent problem of specifying a general scheme. Since it is frequently required to pass strings by reference, chiefly between modules written in the same language, the type specification mechanism permits modules to define private string types. A parameter to a procedure may then be defined as being a reference to such a type.

The mechanism of passing a standard string parameter is as follows:

Two items are pushed on the stack: the first item pushed is an unsigned 32-bit integer specifying the length of the string; the item is a pointer to the first character in the string.

Vectors & Arrays

The passing of vectors and arrays is difficult to standardise, for reasons relating to the specification of the various programming languages. As an example, a two-dimensional array is implemented in Fortran 77 in the reverse dimension order to that used in Pascal, as below:

```
a: ARRAY [1..5, 1..3] OF Integer              -- Pascal

INTEGER a (1:5,1:3)                           -- Fortran 77
```

In Fortran, the element which follows a(2,2) in memory is a(3,2), whereas in Pascal the element a[2,2] is followed by a[2,3].

Parameters

Another difficulty is that in some languages an array is defined by the address of the 0th element (e.g. C) whereas in others the lowest bound of an array may have any value. This leads to variations in the way in which an array may be referenced (e.g. by the address of the 0th element or by the address of the first actual element), to produce the most efficient code.

As a consequence of such problems, this standard does not define the mechanism used to pass (or, where relevant, to return) arrays or vectors. The details of these operations are left to the individual language implementation. However it should be noted that such items may be passed using the standard mechanism, if they form part of a record-type variable. It is permitted to define a record structure in which a vector (i.e. a single-dimension array) occurs as a field. To achieve the effect of multiple-dimensional arrays, the elements of the vector may be defined as records, themselves containing vectors, etc. However this method of working is obviously restricted to fixed-size arrays, since the (maximum size of a record must be known at link-time, according to this standard.

Results

Languages vary in the type and number of results which it is possible to return from a function. It is intended that this standard will cater for the majority of cases in an efficient fashion. In general, results are returned either in registers or via information passed on the stack. Such information is always pushed after all information relating to parameters has been pushed. The result mechanisms have been arranged in such a way that it will generally be possible to call and define functions with multiple results in (suitable systems-type) languages which only permit single results: this is achieved by handling the result information as if it were extra parameter information.

Scalar

If the first or only result of a function is a simple scalar item, it must be returned in:

R0 for a 1..32-bit integer result;
R0 (low) / R1 (high) for a 33..64-bit integer result;
F0 for a single-precision floating point result;
F0 / F1 for a double-precision floating-point result.

In the case of integer results of less than 32 bits, the result is returned in the least significant end of R0. The unused bits of R0 must be zeroed by the called function if the result is to be treated as an unsigned quantity, and made equal to the most significant bit of the actual result if the result is signed. The same rule applies to the unused bits of R1, for the case of results of between 33 and 63 bits.

Non-scalar

If a function result is not a scalar item (e.g. a record or a string), the calling procedure must itself allocate space for the result, and pass the address of this to the called procedure. To return such a result, the called procedure stores the value into this space using the pointer it was passed, then returns. The detailed mechanism for this is as below:

Strings

Where a function returns a first or only result which is a string, the caller must allocate a store buffer into which the function will place the characters of the string immediately before it returns. The calling code must push two items on the stack, after all the parameter information has been pushed. The first item pushed is the size in bytes of the buffer. This is the maximum number of actual characters which may be written into the buffer by the function. The second item pushed is the address of the first byte of this buffer.

When the function is about to return, it must ensure that the number of characters to be returned is less than or equal to the size of the buffer. It is an error unless this condition holds, and this must be immediately indicated to the calling code by signalling. Otherwise the function stores the result into the buffer, loads R0 with the length of the result (i.e. the actual number of characters that were written) as an unsigned doubleword, and returns.

Records and fixed-size array

If a function is defined as returning a record result, the type information for its external symbol must specify, in the result part, how many bytes of information the record contains. This may be an upper bound if the language in which the function is written permits variable-sized records. On the corresponding reference to the function by the calling code, the record size

given must match that of the definition; this will be checked at link time. Hence the size of the result must be known at the calling point. The sequence for calling a record function is then:

The caller allocates a store area to receive the result; the address of this area is pushed (after all parameter information has been pushed) as a single 32-bit pointer; the function is called, and stores its result value into the buffer whose address was passed; the function returns.

Multiple results

Most high level language procedures return either no result or one result. However this standard also defines the mechanism to be used to return any number of results from a function.

Where a function returns more than one result, the second and any further results are returned via pointers as for non-scalar results, even if they are scalar items; only the first result, if it is a is a scalar, is returned in a register or register pair. The handling of extra results depends upon the type of each, as described below.

Scalars, records, and fixed-size arrays

For any extra result whose size is known (including scalars), the caller pushes a single 32-bit pointer which is the address of the (possibly temporary) location to receive it. The function writes the result value to this location, via the pointer, before it returns.

Strings

For extra string results (i.e. any non-first string result) a modified form of the single string result mechanism is used.

Since there may be an arbitrary number of such results, but there are a limited number of registers available, it is not possible in a general way to return the actual length(s) of extra string results in registers. Instead, for each extra which is a string, the calling procedure pushes three 32-bit items: first, the address of a 32-bit unsigned integer to receive the actual length of the result string; then the size of the buffer to receive the result characters; finally, the address of the buffer. Thus the stacked information is as for a first or only string result, with an extra pointer item, pushed first, via which the actual length will be returned.

When returning any string results, a called procedure must first check that no result would overflow the buffer which was provided to contain it. This must be done before any result information at all is written to memory, and if a result would overflow then an appropriate error should immediately be signalled.

Otherwise the procedure should write the characters of each string result into the appropriate buffer, and the length of the result into the corresponding length field whose address was supplied.

Result order

The order in which result information is pushed is right-to-left as for parameters, i.e. information about the last result is pushed first by the

calling code, and information about the first result (if it requires any, e.g. a record result) is pushed last, immediately before the actual call. All result information is pushed after any parameter information.

Call instruction

All procedure calls in this standard are performed by a CXP instruction. This implies (as the definition of AOF states) that all conforming language systems operate under the normal 32000 module mechanism.

Return instruction

Return from a procedure is achieved by the instruction:

```
RXP  N
```

where N is the total number of bytes of parameter and result information which was pushed onto the stack by the calling code before the CXP instruction.

Register use

None of the general registers (R0-R7) or floating-point registers (F0-F7) is required to be preserved across a call. Hence the calling code must ensure that all important information contained in the general and floating-point registers is saved to store before the call is made, and must make no assumptions about the contents of these registers on return (other than those used for results).

The contents of the FP register must be preserved by the called procedure; this is conventionally done through use of the ENTER and EXIT instructions. The PC, SB and MOD registers are automatically preserved through the use of the CXP and RXP instructions. On return from a procedure the SP register must be restored to the state it had before any parameter or result information was pushed, this is achieved by the RXP instruction as described above. The PSR on return is not defined, i.e. no assumptions may be made about its state.

Examples

(1) Called procedure:

```
PROCEDURE  P1 (A, B : INTEGER; S : STRING)
```

Call:

```
P1 (1, X+4, "Hello")
```

Calling code:

```
        MOVQD   5, TOS
        ADDR    text, TOS

        MOVD    X, TOS
        ADDQD   4, TOS

        MOVQD   1, TOS
        CXP     P1


text    DCS     'Hello'
```

(2) Called procedure:

```
PROCEDURE  P2 (VAR X: REAL; Y : LONGREAL; J : INTEGER)
```

Call:

```
P2 (A[k+3], 0.234567, k)
```

Calling code:

```
        MOVD  k, R1
        MOVD  R1, TOS

        MOVL  =0.234567, TOS

        ADDR  A+12[R1:D], TOS

        CXP   P2
```

(3) Called procedure:

```
FUNCTION  F3 (Q : STRING; P, Q : INTEGER) : INTEGER
```

Call:

```
k := F3 (Message, j, 17)
```

Calling code:

```
ADDR    @17, TOS
MOVD    j, TOS
MOVD    Message_length, TOS
ADDR    Message, TOS
CXP     F3
MOVD    R0, k
```

(4) Called procedure:

```
FUNCTION  F4 (I, J : INTEGER) : INTEGER, REAL
```

Call:

```
a, Q[3] := F4 (1, 2)
```

Calling code:

```
MOVQD   2, TOS
MOVQD   1, TOS
ADDR    Q+12, TOS    ; address Q[3] passed for result
CXP     F4
MOVD    R0, a
```

(5) Called procedure:

```
FUNCTION  F5 (R: LONGREAL) : STRING
```

Call:

```
S := F5 (0.2536)
```

Calling code:

```
        MOVL    =0.2536, TOS        ; parameter

        MOVD    =S__size, TOS       ; size of result buffer
        ADDR    S, TOS              ; address of result buffer

        CXP     F5

        MOVD    R0, S_length        ; actual length returned in R0
```

(6) Called procedure:

```
FUNCTION  F6 (S : STRING) : STRING, STRING, STRING, INTEGER
```

Call:

```
s1, s2, s3, status := F6 (Name)
```

Calling code:

```
        MOVD    Name_length, TOS
        ADDR    Name, TOS

        ADDR    status, TOS

        ADDR    s3_length, TOS
        MOVD    =s3__size, TOS
        ADDR    s3, TOS

        ADDR    s2_length, TOS
        MOVD    =s2__size, TOS
        ADDR    s2, TOS

        MOVD    =s1__size, TOS
        ADDR    s1, TOS

        CXP     F6

        MOVD    R0, s1_length
```

Panos Technical

Acorn 32016 Second Processor
Object File Format
Issue A Provisional
Design Document

24-May-84

Object File Format

Contents

Change record
02-May-84 : Roffified.
22-May-84 : Amendments made according to MT's comments.

Overview

An object file is a sequence of one or more independent module descriptions. Each module is in byte stream format: single-byte commands with a variable number of parameters following. The commands are instructions to a linker to allocate and initialise areas of store, and to define the position of global symbols, referenced across modules. Each module description, unless it contains no code, is directly related to a module as defined in the architecture of the machine. It is required that all language systems which make use of this format operate with the 32016 module mechanism.

Overall structure of an object file

The structure of an object file may be summarised as below:

```
module-header-command
module-description-command *
module-end-command
( module-header-command
  module-description-command *
  module-end-command ) *
```

where:

```
X *    means an item comprising 0 or more occurrences of item X
X ?    means an item comprising 0 or 1 occurrences of item X
( X
  Y )  means item X followed by item Y, treated as a single item
```

This definition covers two formats: packed format and general format. Packed format is the standard format for use in Acorn systems; general format allows a more flexible but less efficient layout of the module description. Where possible packed format should always be generated, but in certain cases (e.g. initial compiler development and testing) general format may be used. All system components handling AOF will handle packed format modules; certain components (e.g. the static linker) also accept general format.

In packed format, there is a fixed order in which the module description commands must occur. This order is defined in appendix C.

In general format, module description commands may occur in any order. Comments are also accepted in general format - this may be useful in compiler testing.

Description of Format

Notational Conventions:

nnn — base 10 number - decimal radix is the default

bb_xxxx — base bb number, where bb is in decimal e.g. 16_2F = 47

LL..UU — number range LL to UU (inclusive)

<b> — byte value b - whether it is treated as signed or unsigned is indicated in the context of individual uses within the format. Bit 0 is the least significant bit (LSB), bit 7 is the MSB.

<<v>> — 32-bit signed value, presented as 4 bytes in order LSB..MSB. Bit 0 is the LSB, bit 31 the MSB.

[d] — variable length integer field - this is encoded in the objects stream in a similar way to that in which displacement fields are treated by the 32016 CPU, as follows:

```
One byte is read from the stream.
If bit 7 of the byte is clear then
     the remaining bits (0. .6) form a 7 bit value
else if bit 6 is clear then
     bits (0..5) and the next byte from the stream form a
     14-bit value with the 6 bits from the first byte at the
     most significant end.
else if bit 5 is clear then
     bits (0..4) and the next three bytes from the stream (in
     order MSB..LSB) form a 29-bit value.
else if bits 0..4 are all 0 (zeros)
     the next four bytes in order LSB..MSB from a 32-bit value.
else
     the displacement field is illegal and an error will be
     reported.
```

This type of field will be referred to as a "disp". The interpretation of the value (i.e. whether it is treated as signed or unsigned) is indicated in each context where a disp

field is used. Fields marked as being signed are handled by copying the topmost bit of the original n-bit value into the remaining 32-n bits. In the case of unsigned fields, 0 is copied into these bits. Hence for a single-byte disp field (7-bit, value) the values which may be represented are -64..63 if the field is specified as signed, and 0..127 if unsigned; 2-byte (14-bit) disps encode -8192..8191 when the field is signed, and 0..16383 unsigned, etc.

"sss" byte-string consisting of a length byte (c) followed by C bytes of data (commonly text). A null string is represented by a single 0 byte.

..LL.. byte-sequence consisting of a length field [L] (an unsigned disp), followed by a sequence of L bytes of information. The empty sequence is a disp field, value 0.

< X > conditional field - the occurrence of item X depends on a preceding item's value.

Module header command

o Form:

<1> <<16_32456250>> <type> {[length]} "name" "time" {"source"} "info" {[language]}

o Fields:

Type is an 8-bit set of flags.
Length if present is a unsigned disp field. >= 12.
Name is an identifier string, or null.
Time is a date+time string (format defined below).
Source if present is a filename string (syntax system-specific).
Info is a general text string.
Language if present is an unsigned disp field.

a Effect:

Defines the start of a module. This command comes only at the start of an object file or immediately after a module end command. The magic number is required as a consistency check to reduce the risk of accidents with files which are not object files.

The type field specifies attributes of the module; the bits in this field are as below:

0 : This specifies the format of the module. If bit 0 is 0, the module is in general format; there is no length field. Otherwise the module is in packed format; the length of the module description follows as a disp. The length is the total number of bytes from the first byte of the command to the matching end-module command, inclusive.

1 : Bit 1 specifies the case-sensitivity of the names of global symbols appearing in the module. If bit 1 is clear (0), module and symbol identifiers are treated as case-insensitive; otherwise they are case-sensitive. E.g. C and Modula 2 are normally implemented with identifiers case-sensitive - "Sheila" is not the same as "SHEILA", whereas in Pascal, FORTRAN and IMP the case of an identifier is not significant. Note that the case of other identifiers, in particular area names, is never significant.

2 : This bit may only be set if bit 0 (packed format) is set. It implies that the description of each PIC, non-common area is contiguous,

(i.e. all such areas have type bit 8 set in their declaration), that the start of the data in each corresponding "store block" command is aligned (using NOP or Repeat NOP commands) on a page boundary with respect to the start of the module header command, and that the module description as a whole is an exact multiple of the page size in length. Code generators are NOT expected to produce modules in this aligned mode, since it is rather expensive in disc space: this bit will normally only be set by system utilities which improve the format of frequently referenced modules. The alignment mechanism allows the optimisation of module loading in VM systems so that the data may be paged-in rather than force-loaded.

3 : This bit indicates the presence or absence of the language field. If it is clear there is no language field. If it is set then the language field is present - this contains a number indicating which language generator was used to produce this module. Each language implementation producing AOF is assigned a unique value, for use in this field. This information will be made available (in a manner not defined here) at run time, e.g. for the use of diagnostic and other procedures.

4 : If this bit is 0 then no source field is present. If it is 1 then the source field is present, and should contain the name of the source file from which the current module was generated.

All other bits in the type field are reserved, and must be 0.

Name is an identifier for this module (rules given in appendix A). It may be used by certain languages (e.g. Modula 2) for consistency checks. No two modules used in a linking operation may have the same non-null name. A null name may be used to avoid difficulties in generating unique names.

Time is a string in standard format giving the date and time of generation of the original object file. The syntax of this is specified in appendix B. The string may be null if the date and time of generation are not known.

Info is a text field which should contain the name of the object file generator, the source language name, compiler version etc.

Values for the language field are, as indicated, assigned uniquely to each language implementation. Appendix D gives the current list of languages for which codes are assigned. The correspondence between a particular running module and its generating language will be made available at run time (although this mechanism is not defined here). This is intended to be of use in a general mixed-language programming environment. E.g. a diagnostic routine will wish to determine whether a particular module is written in the language which it is specifically designed to handle.

Module end command

o Form:

<2>

o Effect:

This code marks the end of the current module.

File end command

o Form:

<3>

o Effect:

This command marks the end of significant data in the file. The linker, or other program processing an object file, will not read past this point in the file. Note that if a file end command occurs anywhere other than immediately after a module end command, the object file is assumed to be corrupt and any processing will be abandoned. Hence a compiler may conveniently abort the generation of an object file by dumping this command into the file and stopping. The file end command is optional - if the object file processor encounters the logical end of the object file at a point where it would accept this command, the effect will be the same. For convenience, it is recommended that this command should not therefore be used, other than to abort generation as described above.

Module description commands

The following commands describe the contents of a particular module and may only occur between a module header command and a module-end command.

Declare area

o Form:

<4> <<flags>> <align> [size] "name"

o Fields:

Flags is a 32-bit set of binary flags.
Align is an unsigned byte.
Size is an unsigned disp.
Name is either null or conforms to identifier syntax.

o Effect:

Declares an area of store into which items may be placed by commands in this module description. Areas are numbered (within a module) in order of declaration - the first area declared is area 1 etc. The number an area is given is hereafter termed its tag. Tags are used in other commands in order to refer to specific areas, and are represented in the format as unsigned disp fields.

The individual bits in the flags field have meanings as follows:

0 : If set, the area is read-only - once loaded it must not be modified. If this bit is clear it will be assumed that write access is required to this area.

1 : This bit being set implies that the area is position independent - it may be loaded at any address without any change being required to the contents. In particular, if this bit is set, there must not be any relocation of items in the area, and if the area is a code area, the code must be pure, re-entrant, and make all references to external objects via pointers in other areas (accessed via SB, FP, link table etc).

2 : Setting this bit implies that the linker may arrange for this area to be shared between different processes, i.e. there will only be one copy of the area in physical memory. This requires the area to be position independent, since the virtual address at which it resides may vary between processes. It is also normal only to share read-only areas, but this is not enforced unless the area is a code area (see bit 3 below).

3 : This bit if set marks the code area, if any, in this module. Only one area with this bit set may be declared, since there is only one pointer in the module table to the code area for this module, and hence only one area is directly callable by other modules (by the CXP/CXPD instructions). Note also that if this bit is set, and bit 2 is set then bits 1 and 0 must also be set, i.e. shareable code areas must be position-independent and read-only. If no area in a module is declared as a code area, then the linker will not allocate a module table entry for the module, but will still create and initialise the data areas described (given that the module is loaded, of course).

4 : Common area specification. If this bit is set, this area will be treated as a 'common' area by the linker. That is, all modules specifying this area by name reference the same memory, rather than being allocated individual, private sections of memory. Thus e.g. FORTRAN common blocks will have this bit set. If this bit is set, the name field must be non-null, i.e. the area has a global name. This enables other modules to reference it. This bit must not be set if bit 3 is set, i.e. code areas may not be common.

5 : Common area definition. If this bit is set then bit 4 must be set. It implies that the common area specified is being actually defined by this module, rather than simply referenced by it. The effect of this is to set the size of the area of the specified name. Any module specifying a common block of this name must give a smaller or equal size for it; it is an error if the size is greater. In one link operation, at most one "declare area" command for a particular common area may have this bit set. If none has this bit set, the space allocated to it at link time will be the largest size specified in any contributing module.

6 : No initialisation. This bit is used to indicate to the linker whether there are commands in this module description which initialise the contents of the area being declared. If it is set, there will be no such commands (store block, relocate doubleword, repeat store) for this area - it is an error if any are found. If it is clear, such commands will be accepted and acted upon; it is not an error if there is none, but all code generators should ensure that this bit is set if it can be. This bit must not be set if the 'code' bit is set (i.e. code areas must be initialised in some way!). This facility is provided as an

optimisation so that areas whose initial contents do not matter (e.g. a heap or private stack-area) need not take up space in the image file.

7 : No external relocation. If this bit is set there must be no

"relocate doubleword" commands using modes 3 or 4 in this area. Use of this bit when possible permits the linker (or other program reading the file) to omit a scan of the description of this area to determine the names of external symbols referenced from it.

8 : Contiguous initialisation. If this bit is set, then bits 1 and 7 must also be set, i.e. the area is position independent. This bit is used to indicate that the area is defined as a single contiguous block of data. The definition of the area should consist of one "set position" command, to the start of the area, followed by a single "store block" command, where the length of the data is the same as the length of the area, as given in this declaration. It is an error if this bit is set and the area is not so defined. This bit is provided to permit the optimisation of module loading in certain circumstances - only one transfer is required to set up the whole area.

All other bits in the flag field are reserved, and must be 0.

Align specifies to what boundary the contents of this area should be aligned, as a power of 2. Hence 0 here means alignment to byte boundary (effectively no alignment); 1 means word align, 2 means double-word align, 3 means 8-byte align etc. The maximum value allowed here is defined by the linker (it is currently 10, i.e. 1K).

Size is the size of the area in bytes. The specified amount of space, starting at a suitably aligned address, will be reserved by the linker, whether or not it is all initialised in this module.

Name is a textual name to be associated with this area. It will normally be null unless the "common" bit is set, in which case it must be a valid identifier, so that the area may be referred to from other modules. If the common bit is clear then the name will be ignored. All area declarations in a single linking operation which reference the same common area must specify the same flags (except for the "common area definition" flag). As indicated in the description of the "module header" command, the case of letters in area names is not significant. The linker will arrange that all contributions to a common area are overlaid in the order of module loading. The placement of non-common areas is entirely in the linker's control.

Define global symbol

o Form:

<5> <type> [offset] { [area] } "name" { ..type_info.. }

o Fields:

Type is an unsigned byte.
Offset is a signed disp.
Area if present is the tag of a declared area.

Name is an identifier conforming to the rules below.
Type_info if present is a byte-sequence defining the symbol's use.

o Effect:

Defines "name" as a global symbol. For each global symbol referenced by any module(s), there must be a definition in some module. The type field defines characteristics of the item identified by the symbol, as follows:

0 : Absolute value - area is not present. Offset is the symbol. This is used for global constants.

1 : Data symbol - area is the tag of the area in which the item lies, and offset is the displacement of the item from the start of that area. There is no restriction on the value of offset.

2 : Code symbol - area is not present. Offset is the displacement of the item from the start of the code area in this module (there must be one), and must lie in the range 0..(size of code area-1). Apart from this there is no restriction on the value of offset. Hardware limitations mean that for each module table entry the code items accessible via that entry must lie within a 64K range. However the linker will automatically allocate further module table entries in order to ensure that all code items within a module are accessible, by setting a different program base address into the 2nd (and further) module table entries. This is transparent to object file generators; it is also not likely to occur often in practice - few modules contain over 64K of code! This however means that code generators must use the provided mechanism to generate code-item descriptors - they may not plant code which dynamically constructs these, since the run-time may vary.

128 : Absolute value with type information.

129 : Data symbol with type information.

130 : Code symbol with type information.

For types 128, 129 and 130, the type_info field is present, and gives detailed information about the nature of the item; the format of this field is defined in appendix E. All other values of the type field are reserved.

Defining the symbol "GO" is now equivalent to the 'define entry' command, except that type checking is possible.

Set position

o Form:

<6> [area] [offset]

o Fields:

Area is the tag of a declared area. Offset is a unsigned disp.

o Effect:

The Current Position (CP) is set to point into the image of the area being built. CP has two components - the Current Area, CA, and the Current Offset, CO. This command sets CA to the area parameter and CO to the offset parameter. After the operation, CO must lie in the range 0..size(CA), and CA must lie in the range 1..(number of areas declared in this module). The initial position is area 1, offset 0 (unless there are

no area declarations).

Store block

o Form:

<7> ..data..

o Fields:

Data is a byte-sequence.

o Effect:

The block of data which follows is L bytes long, and is stored into memory at CP. L must lie in the range 0..(size(CA)-CO) . After the block has been stored, L is added to CO.

Repeat store

o Form:

(3> [count] ..data..

o Fields:

Count is an unsigned disp. Data is a byte-sequence.

o Effect:

This command is used when a pattern of data occurs repeatedly within an area in module, e.g. an array of items which are all initialised to the same value. The effect of the command is exactly the same as if 'count' store block commands with the same data had occurred in the file.

Initialise link

o Form:

<9> [ext_no] <mode> \[[offset] \} \[[area] \} \{"module" "name"\}

o Fields:

Mode is an unsigned byte in the range 0..7 (not 2).
Ext_no is a signed disp.
Offset if present is a signed disp.
Area if present is the tag of a declared area.
Module if present is an identifier string, or is null.
Name if present is an identifier string.

o Effect:

This command is used to specify an entry required in the link table for this module. Ext_no is an index into the link table, with the same sense as an 'external number' in instructions, i.e. extno*4 gives the byte offset of the entry from the base of the link table. Note that extno may be negative, i.e. it is possible to have a link table starting before the address specified in the module table entry, and indexed backwards. This permits code generators to be efficient in their use of external references - they may use more byte-sized displacement values for the

external number.

N.B. External entries -1, -2, -3 and -4 are permanently reserved, and must not be initialised by this command.

It should be noted that the allocation of space for the link table is entirely under the control of the linker: the module does not define it itself. The link table's size is worked out on the basis of the minimum and maximum values of extno encountered in "initialise link" commands. As a consequence of this, object file generators are constrained to allocate external numbers contiguously (with the exception of entries -4..-1), and to include entry 0. Hence if there is one external reference, it must be through entry 0; if there are two, they must be via entries -5 and 0, or entries 0 and 1, etc. It is an error if the same entry is initialised more than once.

The mode field specifies which type of initialisation is required for the given entry. In each mode only those optional fields which are relevant are supplied in the command:

0 : the value of the entry is offset, i.e. a constant.

1 : the value of the entry is the address of the start of the area whose tag is supplied, plus the supplied offset.

3 : the value of the entry is the address of the global symbol defined by the name and module supplied, plus offset. Offset is normally, but not necessarily, 0. If the module field is not null, then the entry will only be looked for in the module of that name (there must be one). The global symbol will normally be defined in some other module, but there is no reason why it may not be defined in this module.

4 : the value of the entry is a code entry descriptor for the global symbol identified by the given module and name fields. As for mode 3, the module field may be null, with the same meaning. The symbol must be defined in some module, as a code entry. No offset field is supplied in this mode.

5 : the value of the entry is a code entry descriptor for a local entry point. In this case offset is the displacement of the entry from the start of the code area. The effect of this is as if the entry point had been defined as a global code symbol, and referenced by mode 4 above, except that no external definition need be created for it, nor (more importantly) need its name be quoted. This type of entry may be useful where an external procedure (i.e. one called by CXP), must be called from within this module. This is also relevant to a language such as C, in which all procedures (including so-called 'static', i.e. non-global ones) must be called by the same mechanism. Although this mode has been provided for convenience because there is no "BXP" instruction), compiler writers may care to note that an alternative solution to this problem is to use the instruction sequence:

```
SPRD   MOD, TOS
BSR    proc
```

which simulates a "branch to external procedure" instruction. The latter solution may be more efficient (in space terms) if the procedure concerned is called only once or twice from within the module. Otherwise the link table entry solution is cheaper in space terms, although marginally slower.

6 : the value of the entry is offset + the allocated length of the common area whose tag is given. This will be the largest value which was specified in any declaration of this common area. The tag must be that

of a common area.

7 : the value of the entry is offset + the address of the first byte beyond the space allocated for the common area whose tag is given. This is equivalent to the sum of the values obtained using mode 1 (with offset=0) and mode 6.

Check use

o Form:

<10> <type> "module" "name" ..type_info..

o Fields:

Type is an unsigned byte in the range 0..2.
Module is an identifier, or null.
Name is an identifier.
Type_info is a byte-sequence.

o Effect:

This command is used to check consistency of usage between the definition of a global symbol and a reference to it from this module. It is an error if the symbol named is not of the same basic type (i.e. absolute, data or code), or if the type information (if any) given at the symbol's definition is not compatible with the information specified here. If the length of the type_info field here is 0, then only the basic type is checked. Note that even if this command is not used, the linker will still check consistency to some extent: it is an error if a symbol defined as a data or absolute item is referenced as a code item, and it will issue warnings if a code item is referenced by address rather than by descriptor.

Relocate doubleword

o Form:

<11> <mode> [[offset]] [[area]] ["module" "name"]

o Fields:

Mode is an unsigned byte in the range 0..7 (not 2).
Offset if present is a signed disp.
Area if present is the tag of a declared area.
Module if present is an identifier string, or is null.
Name if present is an identifier string.

o Effect:

This command is used to statically relocate a doubleword in store. It is provided for use by language implementations which require that a pointer may be initialised to refer to some other item in store. It is similar to the "Initialise link" command: the meanings of the fields are identical but the effect is to initialise the double word pointed at by CP, rather than a specified entry in the link table. See the description of that command for a full specification of the modes of initialisation possible. This command must not be used within an area which is declared

as being position independent or shareable; in fact its use is strongly discouraged, since it complicates the overall clean architecture of the system. It should only be used where there is no other practical solution (for example dynamic initialisation). When this command is met, it is an error unless CO is in the range 0..(size (CA)-4). After this command is executed. CO is stepped on by 4.

NOP

o Form:

<12>

o Effect:

None. This command does nothing, but its presence is required to satisfy certain constraints of packed format, in that parts of a packed format file may require to be aligned on particular boundaries.

Repeat NOP

o Form:

<13> ..ignored..

o Fields:

Ignored is a byte-sequence.

o Effect:

This command is an extension of NOP, with the effect of making the program reading the file ignore a number of following bytes. It is meant to save time on the part of object file generators and the linker, in that they do not have to write (or interpret) a large number of NOPs, but just skip that section of the file.

Comment

o Form:

<14> "comment"

o Fields:

Comment is a string of bytes

o Effect:

This command is used to allow abject file generators to dump information which may be of use for debugging purposes. For example a

compiler may dump the line number in the source file of the statement which generated the following code for use by a program which correlates code with source. No restriction is placed on the contents of the comment string - it will be completely ignored by the linker. Comments may not occur in packed format.

Define SB

o Form:

<15> <mode> { [area] } [offset] { "module" "name" }

o Fields:

Mode is an unsigned byte in the range 0..3 (not 2).
Area if present is the tag of a declared area.
Offset is a signed disp.
Module if present is an identifier or null.
Name if present is an identifier.

o Effect:

This command tells the linker how the static base field in the module descriptor(s) for this module should be initialised. This command may occur at most once in any module, unless there is no code area in this module, in which case it must not occur at all. If there is a code area, but this command is not given, then the linker will initialise the SB to point at an inaccessible area of store (if possible). The value to which the SB will be set is determined by the value of mode, as follows:

0 : SB is set to the constant value given by offset. Area is not present. This might be useful for example where a device handler module accesses fixed-address memory-mapped device registers, in that the static base could conveniently point at the start of these.

1 : SB is set to offset + the start address, of the area whose tag is given. This is similar to mode 1 in the "initialise link" command.

3 : SB is set to the offset + the value of the global symbol whose name (and perhaps module name) is given.

All other values of the mode field are reserved.

Define entry

o Form:

<16> [offset]

o Fields:

Offset is an unsigned disp.

o Effect:

Defines the global start address for the program. Offset is the displacement from the start of the code area of the first instruction to be executed when the program is run. This command may be used at most

once in a module, and exactly one of the modules bound together in a linking operation must contain this command. It is an error if the current module declares no code, or offset is not in the range 0..(size(code area)-1).

This command is now obsolete, but will be retained for reasons of compatibility. The preferred way to mark the entry point is to define a global code symbol "GO" (see the description of the 'define global symbol' command).

Define handler

o Form:

<17> [offset]

o Fields:

Offset is an unsigned disp.

o Effect:

Defines the address of a module-local "handler" routine which will be called prior to execution of this module, on termination of the program, and also in the event of synchronous errors (e.g. divide by zero, illegal instruction trap) occurring while an instance of this module is active. Offset is the displacement from the start of the code area of the first instruction to be executed in the handler; it must lie in the range 0..size(code area)-1.

A handler is defined as a piece of code local to a module (rather than a general code address), so that, should an error occur during handler execution, the environment is correct as far as a second-level handler is concerned, i.e. the cause of the problem is traceable to the actual module whose handler was originally called. In many cases the local handler will be a very simple, short procedure which calls (via CXP) a larger procedure specific to the language in which the module was written. Thus the varying requirements of languages in respect of initialisation and error handling may be met in a standard fashion. For further details of the specification of handler routines, see the appropriate document describing the runtime support system.

Define Type Dictionary

o Form:

<18> ..type_information..

o Fields:

type_information is a byte sequence

o Effect:

Defines local (tagged) and global (named) type information used within the module.

In the above definitions, an identifier must conform to the following rules:

The character set is ASCII. All printing characters may be used, including space, i.e. the range of valid character values is 32..126. An identifier comprises between 1 and 255 characters selected from this range without restriction. In respect of case-sensitivity (i.e. whether "a" = "A"), the fallowing rules apply:

If the 'case-sensitive' flag in a module header is set, then all module and global symbol identifiers used in that module are taken to be case-sensitive, i.e. "Fred" is distinct from "FRED". Otherwise all names in the module are preserved exactly as they were generated, but they are marked as case-insensitive. When two strings are to be compared, the flags of both names are tested, and only if both specify case-insensitivity is the comparison case-insensitive (i.e. the identifiers are reduced to the same case before comparison). Otherwise case must match precisely for the equality test to succeed.

Appendix B: Syntax of date and time strings

The format of a date+time string is:

```
     "YYYY-MM-DD HH:MM:SS"
or   "YYYY-MM-DD HH:MM:SS.CC"
```

where:

YYYY is the year as 4 digits AD.

MM is the month of the year in the range "01" .. "12"

DD is the day of the month in the range "01" .. "31"

HH is the hour in the range "00" .. "23"

MM is the minute in the range "00" .. "59"

SS is the second in the range "00" .. "59"

CC is the centisecond in the range "00" .. "99"

The centisecond field (i.e. ".CC") is optional, but should be included if possible, to provide greater precision.

Appendix C: Packed Format

In packed format, the order in which commands may occur restricted. The layout of a packed format module description is:

```
module-header              -- the type field has the packed-format
                           -- bit set, and the size field is present
define-entry ?             -- if this module defines the entry point
                           -- using the old mechanism
define-handler ?           -- if this module defines a handler entry
define-SB ?                -- if SB is defined for this module
define-type-dictionary ?   -- if one is required
define-global-symbol *     -- any code symbol definitions (types 2, 130)
                           -- "$GO$" must be defined first, if at all
define-global-symbol *     -- any data or absolute definitions
```

```
init-link/check-use *       -- any initialise link commands using
                            -- mode 4 (external code descriptor) must
                            -- occur here; any check-use command for a
                            -- particular symbol should occur immediately
                            -- after the corresponding "init-link" command
init-link/check-use *       -- commands using all other modes here - again
                            -- with any "check-use" command following the
                            -- "init-link" corresponding to it
declare-area *              -- areas with external relocation (i.e.
                            -- containing 'relocate doubleword'
                            -- commands using modes 3 or 4)
declare-area ?              -- code area, if there is one and it has  _
                            -- not been declared already
declare-area ?              -- area for SB, if SB is defined using mode
                            -- 1, and area has not already been declared
declare-area *              -- all other areas
area-description *          -- area descriptions, in the same order in
                            -- which they were declared.
module-end
```


Appendix D: Language Codes

Current language implementations and codes have been assigned as below:

Language Implementation	Code
"Unknown"	0
Algol 68C	1
BCPL	2
C (1)	3
C (2)	4
Cobol	5
Forth	6
FORTRAN-77	7
IMP	8
Lisp	9
Modula-2	10
Pascal	11

Note:

The "unknown" code is for use where conformance to the Acorn 32016 inter-

language calling standards is not under the control of the specific language generator. Hence e.g. assemblers (in particular the ZASM 32016 assembler) will use this code, although they may provide the facility to specify a language code, if a given piece of code is intended to interface directly to a known language.

N.B. THESE CODES ARE ASSIGNED SOLELY BY ACORN COMPUTERS LTD.

Appendix E: Type definition format

This appendix defines the format of type-description information which should be generated by compilers to indicate the type of all the external items defined in, or referenced from a module.

Syntax of descriptions:

```
{:X:}          descriptor for type X
[V]            unsigned integer disp field
"sss"          byte-string field

? tag1.        alternative: selected on initial unsigned integer disp field
   f1          having value 0 for case 1, 1 for case 2 etc.
/ tag2.        tag1, tag2 ... are descriptive text for each alternative
   f2          f1, f2 ... are the definitions of the alternative structures
...
?

X * exp        multiple field - 'exp' occurrences of item X

! comment text
```

Descriptions:

```
!  Padding - this may be used (typically) in record descriptions for the
!  case where fields are aligned internally in the record

    Padding    =    [0]
```

```
                    [number of bits of padding]

!  Lowest level of definition - used where the interpretation of a data item
!  cannot be given - e.g. BCPL "words", which have no implicit type

    Raw-binary =    [1]
                    [number of bits in item]

!  Standard-type string - passed by value, or returned as a function result

    String     =    [2]

!  General integer

    Gen-Int    =    [3]
                    ? unsigned
                    / signed
                    ?
                    [number of bits in integer]
```



```
!  Standard machine floating point types

    Floating   =    [4]
                    ? floating.      ! 32 bits
                    / long.          ! 64 bits
                    ?

! Conventional pre-defined types - these are essentially abbreviations for
! (and hence compatible with) the corresponding general integer type above

    S-Int      =    [5]   ! 32-bit signed integer
    S-Short    =    [6]   ! 16-bit signed integer
    S-Byte     =    [7]   ! 8-bit signed integer
    U-Int      =    [8]   ! 32-bit unsigned integer
    U-Short    =    [9]   ! 16-bit unsigned integer
    U-Byte     =    [10]  ! 8-bit unsigned integer

! compound types

    Vector     =    [11]            ! single-dimension contiguous array
                    ? fixed bounds.
                       [lo bound]
                       [hi bound]
                    / variable bounds.
                    ?
                    {:element type:}

    Array      =    [12]            ! multi-dimension or non-contiguous
                    [number of dimensions]
                    ? fixed bounds.
```

```
                    !  bounds below as:  A (L1..H1, L2..H2, L3..H3 etc)
                    ([lo bound] [hi bound]) * number of dimensions
                  / variable bounds.
                  ?


    Record      =   [13]
                  ? untyped.
                    [size in bytes]
                  / typed
                    [number of fields]
                    {:field type:} * number of fields
                  / named + typed
                    [number of fields]
                    ("field name" {:field type:} ) * number of fields
                  ?


    Variant     =   [14]
                  [number of alternatives]
                  {:alternative type:} * number of alternatives
```



```
    Restricted  =   [15]
                  {:integer type:} [lo bound] [hi bound]
                  ? subrange.
                  / enumerated.
                    ? weak definition.
                    / strong definition.
                      "element-name" * (hi bound - lo bound + 1)
                    ?
                  ?


    Pointer     =   [16]
                  {:referenced type:}


    Named-type  =   [17]
                  "type-name"
                  {:definition:}


    Name-ref    =   [18]
                  "type-name"

    Tagged-type =   [25]
                  [module-local tag]
                  {:definition:}

    Tag-ref     =   [26]
                  [module-local-tag]


! Standard procedure

    Procedure   =   [19]
                  [number of results]
                  {:result type:} * number of results
                  [number of arguments]
                  {:argument type:} * number of arguments


! Flexible procedure i.e. one that takes a variable number of parameters.
```

```
! These are removed by the calling code and not by the called procedure.
! This is an extension and is intended for use by C. Flexible procedures are
! compatible only if the result types are compatible; they are not
! compatible with any other procedures.

    Flex-proc  =   [20]
                   {:result type:}


! General untyped pointer - e.g. ADDRESS type in MODULA-2

    Address    =   [21]
```



```
! Nil parameter - for use by F77 when no formal/actual parameter passed: F77
! code conventions require that 1 doubleword is always pushed before the
! CXP. Nil is defined to be a doubleword with value 0, for this purpose.

    Nil        =   [22]

! Language-specific type codes: Private requires that the called procedure
! must be in the same language as this one.  Non-standard is a general
! extension. The semantics of these two codes is not yet defined!!!

    Private    =   [23]
                   [language-code]          ! must match
                   [type-desc-length]
                   {:language-specific type description:}

    Non-standard = [24]
                   [type-code]
                   [type-desc-length]
                   {:non-standard type description:}
```

Acorn 32016 Second Processor
Calling PANOS Standard Procedures from C

17-MAR-86, ADC

Calling PANOS Standard Procedures from C

This document contains the information required to write C programs which call the operating system procedures described in the Panos Programmer's Reference Manual; it explains how the pseudo-language used in that document to describe Panos procedures corresponds to the functions and data types available in C.

Note that some Panos operations are also provided by C library functions which do not need to be called in a special way: for example, getenv() can be used to get the value of a Panos global string variable, system() calls the Panos

command line interpreter to process a command string, exit() calls Stop, and malloc() performs storage allocation.

Declaring Panos Procedures

A C program must contain explicit declarations of all external functions it calls which are not written in C; this includes all of the Panos standard procedures, which are written in Modula-2.

Declarations of non-C functions must contain the keyword asm, as shown in the examples below. This is important: unpredictable run time errors will occur if the asm keyword is omitted.

```
extern void XSelectInput() asm;    /* no result */
extern int SelectInput() asm;      /* int result */
extern DeleteFile() asm;           /* int by default */
```

Data Types

The correspondence between the data types used in the Panos Programmer's Reference Manual (INTEGER, CARDINAL, STRING and so on) and the data types available in C is shown in the table below, and illustrated in the example sections which follow.

Panos Type	C Type
INTEGER	int
CARDINAL	unsigned int
ADDRESS	any pointer type, e.g. (char *), or int
BOOLEAN	int (0 is FALSE, 1 is TRUE)
HIDDEN	int
RECORD(format)	any struct type
STRING	see section on String Parameters below
PROCEDURE	see section on Procedure Parameters below

Error Detection

Most Panos standard procedures are functions which return an integer status code indicating whether or not the requested operation was performed successfully. A negative status code indicates failure (the value is a standard Panos error number); a zero or positive value indicates success.

Panos procedures whose names begin with the letter X (e.g. XFindInput) handle error conditions differently. If an X procedure returns to its caller the operation was successful. If an error occurs, control does not return to the caller: instead, a Panos exception condition is signalled. Conditions can be trapped under program control using the standard procedures described in Section 11 of the Panos Programmer's Reference Manual (Condition Handlers). Conditions which are not trapped cause output of a diagnostic traceback showing the state of the program at the time of the error; execution is then halted.

Either error handling technique can be used in C programs: the programmer can choose whichever is more convenient (status code or signal).

Simple Examples

The easiest functions to call are those with integer-like arguments and results. The C program below uses Panos to 10 random numbers.

```
extern unsigned int Random() asm;  /* obligatory */

main()
{
   unsigned int r;    /* i.e. CARDINAL */
   int i;

   for (i = 0; i < 10; i++) {
      r = Random();
      printf("%u\n", r);
   }
}
```

If you prefer, this can be tidied up a bit by using C's typedef feature to define the type CARDINAL as equivalent to (unsigned int).

```
typedef unsigned int CARDINAL;
extern CARDINAL Random() asm;

main()
{
   CARDINAL r;
   int i;
   for (i = 0; i < 10; i++) {
      r = Random();
      printf("%u\n", r);
   }
}
```

Multiple Results

Some Panos functions return more than one result, e.g. the Allocate function returns a status code and a pointer to the block of storage allocated.

```
Allocate(INTEGER:Size);
              INTEGER: Result
              ADDRESS: BlockPointer
```


In fact, only the first result (INTEGER:Result above) is returned as a proper C function result. For all the others the calling routine must pass pointers to variables where the extra results will be stored. These extra arguments go at the beginning of the argument list, so the function synopsis for Allocate looks like this:

```
typedef char *ADDRESS;  /* a pointer */

int Allocate(BlockPointer, Size)
ADDRESS *BlockPointer;
int Size;
```

You could write a version of the C library function malloc() based on Allocate.

```
extern int Allocate() asm;  /* returns status code */

char *malloc(nbytes)
{
   char *block;   /* points to allocated space */
   if (Allocate(&block, nbytes) < 0)   /* failed */
      return(0);
   else
      return(block);
}
```

Multiple extra arguments are written left to right, in the order they appear in the Panos procedure definition. (Including STRINGS and RECORDS, see below). For example, the OSByte procedure returns four results. Its definition is

```
OSByte(CARDINAL:ByteNo
       CARDINAL:Param1
       CARDINAL:Param2);  INTEGER:Result
                          CARDINAL:Result1
                          CARDINAL:Result2
                          BOOLEAN:CBit
```

In C this becomes

```
int OSByte(Result1, Result2, Cbit, ByteNo, Param1, Param2)
unsigned int *Result1, *Result2, ByteNo, Param1 Param2;
int *Cbit;
```

STRING Parameters

Many Panos procedures require STRING arguments, or return STRING results. Unfortunately, a Panos STRING is different from a C string. C strings are represented by a (char *) pointer to an array of characters, terminated by a NUL ('\0') character. A Panos STRING is represented by two values: a (char *) pointer to an array of characters, and an integer which is the length of the string.

This means that to call a Panos routine with a STRING argument a C program must pass two actual arguments: a pointer to the string, and the length of the string (obtained using the C library function strlen).

The example program below uses the Panos XDeleteFile routine to delete a list of files whose names are read in from the standard input stream.

```
extern void XDeleteFile() asm;      /* no result */

main()
{
   char fname[64];
   while (gets(fname))              /* read file name */
      XDeleteFile(fname,            /* pointer to string */
                  strlen(fname));   /* length at string */
}
```

This program is flawed: it will give a diagnostic traceback and stop if it fails to delete a file (perhaps because the file did not exist). The program below corrects this problem. It calls DeleteFile rather than XDeleteFile, and checks the status code returned by Panos to see if the file was successfully deleted.

```
extern int DeleteFile() asm;        /* returns status code */

main()
{
   char fname[64];
   while (gets(fname)) {
      if (DeleteFile(fname, strlen(fname)) < 0)
         printf("can't delete %s\n", fname);
   }
}
```

STRING Results

Some Panos procedures return STRING results. Each STRING result requires three extra function arguments: a pointer to a buffer where the result string is to be written, an integer indicating the size of this buffer, and a pointer to an integer variable which will be set to the actual length of the string. The extra arguments go before normal arguments.

The program below prints out the current operating system version number by reading the contents of the Panos global string variable SYS$Version using the GetGlobalString "procedure, defined as

```
GetGlobalString(STRING:GlobalStringName);
          INTEGER:Result
          STRING:GlobalStringValue
```

Here is the program.

```
extern int GetGlobalString() asm,
          Stop() asm;

#define MAXLEN 128

main()
{
char buf [MAXLEN];      /* buffer for result string */
int len;                /* actual length of string */
int status;             /* Panos error code */
```



```
if ((status =
     GetGlobalString(buf, MAXLEN, &len,  /* GlobalStringValue */
                     "SYS$Version", 11), /* GlobalStringName */
     < 0)               /* failed */
     Stop(status);
buf[len] = '\0';        /* convert buf to C string */

printf("Panos version: %s\n", buf);
}
```

There are several points to note about this example.

First, the arguments corresponding to the STRING result (GlobalStringValue) go before the normal arguments (GlobalStringName).

Second, a constant string input argument ("SYS$Version") is passed as a C string literal (the C compiler generates a pointer to the first character of the string) and a constant length count (11).

Third, the GlobalStringValue returned by Panos in the array "buf" is not automatically terminated by a '\0' character. Instead, the "len" argument is assigned the actual length of the string value returned. So before using "buf" as a normal C string (by passing it to printf), the program must write a '\0' character at the end of the string.

Beware: there is an important special case of STRING results, where the STRING is the first (or only) result of a Panos function. In this case, only the string result buffer and maximum length count are passed to the procedure as extra arguments. The actual length is returned as the integer result of the function. For example, the program above could be rewritten to call XGetGlobalString rather than GetGlobalString. The definition of XGetGlobalString is

```
XGetGlobalString(STRING:GlobalStringName);
          STRING:GlobalStringValue
```

The new program would look like this.

```
extern int XGetGlobalString() asm;
```

```
#define MAXLEN 128

main()
{
   char buf[MAXLEN];  /* buffer for result string */
   int len;           /* actual length of string */
   int status;        /* Panos error code */

   len = XGetGlobalString(buf, MAXLEN,   /* GlobalStringValue */
                    "SYS$version", 11);  /* GlobalStringName */

   buf[len] = '\0';  /* convert buf to C string */

   printf("Panos version: %s\n", buf);
}
```


RECORD Parameters

RECORD parameters correspond to C struct types. For each RECORD argument the caller must pass a pointer to an appropriate struct variable. RECORD results are handled as for other data types: the caller passes pointers to variables where the results will be stored; these extra arguments go before the normal arguments.

The example below uses GetFileInformation to examine a file whose name is given on the command line.

```
extern int GetFileInformation() asm,
           TextualTimeOfBinaryTime() asm;
extern void Stop() asm;

main(argc, argv)
int argc;
char *argv[];
{
   struct BTim { unsigned int Low, High } date;

   struct FileData {
      unsigned int load_addr,        /* load address */
                   exec_addr,        /* execution address */
                   len,              /* file length */
                   attr;             /* BBC filesys attribs */
   } cat;

   char s[128];         /* string result buffer */
   int len, status;

   if (status =
       GetFileInformation(&cat, &date,  /* results */
                          argv[1],  strlen(argv[1])) < 0
      Stop(status);

   if (status =
       TextualTimeOfBinaryTime(s, 128, &len, &date) < 0)
      Stop(status);
   s[len] = '\0';

   printf("%s: %d bytes, (%s)\n", argv[1], cat.len, s);
}
```

PROCEDURE Parameters

Panos PROCEDURE type parameters correspond to C function pointers.

Consider the Panos ArgumentInit procedure.

```
ArgumentInit(STRING:KeyString
             BOOLEAN:InputWanted
             BOOLEAN:OutputWanted
             STRING:Identification
             PROCEDURE:HelpProcedure);
                         INTEGER:Result
                         HIDDEN:Handle
```


In C, this becomes

```
typedef int BOOLEAN;
#define TRUE  1
#define FALSE 0

int ArgumentInit(Handle, KeyString, KeyStringLen, InputWanted,
             OutputWanted, Id, IdLen, Help)
int *Handle, *KeyStringLen, *IdLen;
BOOLEAN InputWanted, OutputWanted;
char *KeyString, *Id;
void (*Help)();
```

An example using ArgumentInit is given below.

```
#include <stdio.h>

extern int ArgumentInit() asm;
extern void Stop();
typedef int HIDDEN;
#define TRUE 1

static char keystring[] = "Source/A/E-roff",
            ident[] = "roff v1.2";

main()
{
   HIDDEN handle;
   extern void help();
   int status;

   if ((status =
        ArgumentInit(&handle,
                     keystring, strlen(keystring),
                     TRUE, TRUE,
                     ident, strlen(ident),
                     &help)) < 0)
      Stop(status);

   .
   .
   .
}

help()
{
   fprintf(stderr, "roff text formatter\n");
}
```

Acorn 32016 Second Processor
Pandora Definition
Issue B Provisional
User Document

24-Oct-84

Pandora Definition

Contents

Change record

07-Feb-84 : Adjusted for version 5.00 Kernels.
03-Apr-84 : Adjustments by John C Moore to tidy up.

15-Aug-84 : Further changes by John.

04-Oct-84 : Amended by Simone to Pandora Definition Issue B.
All previous references of Tiny Kernel ==> Pandora
TK ==> P

1. Introduction

The main purpose of Pandora is to allow programs running an the 32016 second processor to have access to the BBC operating system. It does this by generating the necessary interface to the TUBE via which the BBC MOST and 32016 SLAVE communicate.

Pandora enables the BBC machine to be used as an I/O processor without requiring the user to have any knowledge of the low-level tube interface protocols. Pandora takes over the hardware Dispatch table and filters out all interrupts pertaining to the TUBE before passing the remainder on to the user via the "Virtual" Dispatch table (see section 2.3).

Certain Tube interrupts correspond to EVENTS signalled by the I/O processor. Events of this type include the timer event and the buffer full event. All these events are enabled explicitly via the OS_BYTE request to the BBC MOS.

There is another class of EVENTS which do not directly correspond to OS_BYTE requests; events of this type are enabled using the OS_CONTROL call. For example one may request that an event is received every time the escape flag is updated (see section 2.2). In addition, this class includes events which cannot be attributed to a specific request to the Pandora. For example, Pandora is unable to determine the originating OS_WRCH request when the host signals a "Can't extend error" whilst spooling. There is an additional problem because sometimes I/O errors can be incorrectly attributed to an otherwise error free SVC call. To avoid this problem spooling should be avoided.

The following section describes in detail how to use Pandora to gain access to the functionality of the BBC microcomputer.

Section 3 describes entry to the control program.

2. User Interface to Pandora

The requestor calls the Pandora by executing a SVC instruction. For direct calls, the byte following the SVC instruction is the SVC number. Whereas, for indirect calls the byte following has the value 16_00 and the SVC number is contained in R0(B). If the SVC number is a valid one, then Pandora passes the request onto the BBC HOST.

Pandora is also entered when the NS32016 signals an exception condition. Pandora is entered via the NS32016 Dispatch table and deals with any exceptions relevant to the operation of the TUBE. Other exceptions are passed onto the control program, control indirecting via the Virtual Dispatch Table.

There are four classes of SVC calls:-

(i) MOS REQUESTS
Calls to request an operation by the host I/O processor.

(ii) EVENT CONTROL CALLS
Calls to mechanise the enabling, disabling and receiving of events.

(iii) VIRTUAL DISPATCH TABLE MANAGEMENT CALL
A call to define the position of the Virtual Dispatch Table.

(iv) MISCELLANEOUS
Calls to control kernel operation.

The MOS REQUEST SVC calls are described in detail in Section 2.1. The EVENT CONTROL calls are presented in Section 2.2 and, in Section 2.3 the Virtual Dispatch Table is discussed together with the associated SVC call. The miscellaneous calls are described in section 2.4.

For easy reference the various SVC calls are listed in Appendix A. In this

document the following conventions apply:

- Numbers not in decimal are prefixed by their base. For example 16_1A is decimal 26.

- Characters are presented in ASCII code (see P 492 of BBC User Guide).

2.1 MOS REQUESTS

The control program gains access to the functionality of the host I/O processor by putting the appropriate values in the NS32016 registers and then executing a SVC instruction.

For several calls to Pandora the requestor is required to pass pointers to data structures. For example, OS_STRING requires that R1 should point at the string to be output. Pandora assumes that all data structures (including the byte following the SVC instruction) reside within the supervisor space. Of course, when no MMU is present then both supervisor and user addresses are the same and therefore this restriction is not visible.

In the description of the operating calls which follow, all registers and flags are preserved unless otherwise stated.

An error is indicated by a return with the F bit set, in which case R1(D) contains the error number. Error numbers in the range -1 to -256 are returned when an error is detected on the I/O processor, (i.e. the least significant byte of R1 is the byte error code returned by the host and all other bits of R1(D) are set. Pandora performs certain checks itself and if it detects an error then it returns an error value in the range 256<=R1(D)<=511.

It should be noted that some of the requests may cause unattributable errors to occur. Far example, a OS_WRCH whilst spooling can result in a "Can't extend error" which may cause an EVENT (see Section 2.2).

RESERVED SVCs
SVCs 0-127 are reserved for use by Acorn.

2.1.1 OS_WRCH

- SVC code = 1
- Effect:

  Write character to currently selected output stream(3) selected with OS_BYTE call 03.
- On Call:

  R1(B) contains character to be written.
- On Return:

  F bit clear always. All registers unchanged.

2.1.2 OS_STRING

- SVC code = 2
- Effect:

  Write character string to the currently selected output stream(s).
- On Call:

  R1(D) is a pointer to a string of characters.
  R2(D) is length.
- On Return:

  F bit clear always. All registers unchanged.

2.1.3 OS_ASCI

- SVC code = 3
- Effect:

  Write character to currently selected output stream(s). If character is a carriage return (ASCII 16_0D) then a line feed (ASCII 16_0A) followed by the carriage return is output.

- On Call:

  R1(B) contains character to be written.

- On Return:

  F bit clear always. All registers unchanged.

2.1.4 OS_NEWL

- SVC code = 4
- Effect:

  Writes the current newline string to the output stream(s), (see OS_SETVDU for details)

- On Call:

  No call parameters.

- On Return:

  F bit clear always. All registers unchanged.

2.1.5 OS_INLINE

- SVC code = 21
- Effect:

Output an inline string with effect as OS_STRING.

- On Call:

  A byte string of format <length><string to be output> follows the SVC instruction in store.

- On Return:

  F bit clear always. All registers unchanged.

2.1.6 OS_RDCH

- SVC code = 5
- Effect:

  Reads a character from the currently selected input stream. The input stream can be selected with OS_BYTE 16_02.

- On Call:

  No call parameters.

- On Return:

  If F bit clear then the read was successful and R1(D) contains the character read (in least significant byte). If F bit set then error or escape has been detected and R1(D) contains error code (=16_11 for escape).

  Notes: If an escape condition is detected than it must be acknowledged using an OS_BYTE call 16_7E.

2.1.7 OS_BYTE

- SVC code = 6
- Effect:

  Invokes a miscellaneous set of operating system calls on the host (the OSBYTE or FX commands of the BBC Microcomputer).

  The mapping between the osbyte calls described in the BBC machine user guide and their Pandora counterparts is:

- On call:

  The call function (6502 A register) is placed in R1(B).

  Parameter 1 (6502 X register) is placed in R2(B).

  Parameter 2 (6502 Y register) is placed in R3(B).

o On return:

The F bit is clear if no error occurred. The validity flag (6502 C flag) is placed into the C bit.

Calls type 0-16_7F return:

R2(D) contains zero extended result (6502 X register).

Calls type 16_80-16_FF return:

R2(D) contains result 1 (6502 X register) in b0-b7 and result 2 (6502 Y register) in b8-b15.

R3(D) contains zero extended result 2 (6502 Y register).

Exceptions to this rule are:

Call 16_82 - R2(D)=0 (indicating client, not host, memory)

Call 16_83 - R2(D)=Lowest address not allocated for use by Pandora (see sections 3, 4).

Call 16_84 - R2(D)=Address of the first byte of Pandora high memory workspace.

Call 16_9D - changes no registers.

2.1.8 OS_WORD

o SVC code = 7

o Effect:

Invokes a miscellaneous set of operating system calls on the MOST. Unlike OS_BYTE calls these calls all require a control block.

On entry to the routine R1(B) contains the OS_WORD function required. R2(D) points to the control block.

A list of OSWORD calls available is in Appendix B. Note that the format of the control block for OSWORD 0 is different to that used when calling OSWORD on the BBC MOST.

2.1.9 OS_CLI

- SVC code = 8
- Effect:

  Sends a line to the BBC mos command interpreter in the HOST.

- On Call:

  R1(D) points at command line. R2(D) is length.

- On Return:

  If F bit clear, command executed successfully and all registers are as on call.

  If F bit set, error detected whilst executing command. R1(D) contains error number. Error messages can be read using OS_ERROR. If the error code is 16_104 then LOADs from within OS_CLI have been prohibited and this call attempted such a LOAD (see section 2.4.6).

  If a program to be loaded and run has a valid 32016 header then the program is entered in supervisor mode as a control program. Otherwise it is entered with the same status (PSR) as caller.

2.1.10 OS_FILE

- SVC code = 13
- Effect:

  Loads and saves data to a file and reads/alters catalogue information.

- On Call:

  R1(B) contains a OS_FILE function number (see below)
  R2(D) points to file/directory name
  R3(D) length of name
  R4(D) contains load address
  R5(D) contains execution address
  R6(D) contains the Data start address or length
  R7(D) contains the Data end address or attributes

- On Return:

  F bit clear: successful request
  register contents depend on OS_FILE function

  F bit set: error detected by HOST whilst executing command
  R1(D) - error number returned by I/O processor (HOST).
  Error message can be read using OS_ERROR
  all other registers unchanged

  OS_FILE
  FUNCTION

  16_FF LOAD file to given address/file's address.
  The low byte of the execution address (R5B) determines
  whether the file should be loaded to it's own address
  (if non-zero) to be the supplied load address (if zero).
  The file's catalogue information will be written into the
  registers, see item 5.

  16_00 SAVE data to file (no wild cards are allowed in the file name).
  If start address = end address then the data is of zero length.

The file's catalogue information will be written into the registers, see item 5.

16_01 WRITE catalogue information for file (no wild cards are allowed in file name).
Length information is not alterable.

16_02 WRITE the load address to the catalogue information.
Only the Load address is required.

16_03 WRITE the execution address to the catalogue information.
Only the execution address is required.

16_04 WRITE the attributes to the catalogue information.
Only the attributes are required.

16_05 READ file's catalogue information.
The Load address (R4 D), Execution address (R3 D), Length in bytes (R6 D), Attributes (R7 D) and Type (in R1 D) are returned for the particular file.

16_06 DELETE the file (no wild cards are allowed in the file name) and return catalogue information.

The attribute of a file is a 4 byte (32 bit) item whose bits refer to the state of various protection flags, and other catalogue information. Filing systems may not use same of the flags.

The bottom 8 bits have the following meanings:

bit	meaning
7	
6	The file is executable by other users.
5	The file is writeable by other users.
4	The file is readable by other users.
3	The file is locked.
2	The file is executable by you.
1	The file is writeable by you.
0	The file is readable by you.

Filing systems with date information available place it in the remaining 24 bits (or a subset of them). Otherwise, they should be returned as 0.

The Type in R1(D) describes the object which was found:

Type 0 is no object found
Type 1 is file found
Type 2 is directory found

2.1.11 OS_FIND

- SVC code = 11
- Effect:

  Opens a file for reading/writing/update.
- On Call:

  R1(B) determines the type of OS_FIND operation (see below).
- On Return:

  F bit clear: successful request, register contents depend on OS_FIND request
  F bit set error detected by HOST while executing command.
  R1(D) contains error number returned by I/O processor.
  error message can be read using OS_ERROR
  all other registers unchanged.
- On Call R1(B) nonzero

  R1(B)<>0 causes a named file to be opened.
  R2(D) points to the name of the file to be opened.
  R3(D) is the length of the file name pointed at by R2.

  If R1(B) = 16_40 the file is opened for input only.
  If R1(B) = 16_80 the file is opened for output only.
  If R1(B) = 16_C0 the file is opened for reading and updating.
- On Exit:

  F bit clear
  implies that the file was successfully opened.
  R1(D) is handle for the opened file
  (which will be in the range 1..255)

  F bit set
  signals an error
  R1(D) = #X100 implies that the file could not be found.
- On call R1(B) zero

  R1(B) = 16_00 causes (a file)/(files) to be closed.

  If R2(B)<>0 the file whose handle is given by R2(S) is closed.
  If R2(B) = 0 all open files are closed
  (including SPOOL and EXEC files).
- On exit:

  F bit clear operation was successful

F bit set operation failed

2.1.12 OS_ARGS

- SCV code = 12
- Effect:

Reads/Writes an OPEN file's attributes.

- On Call:

R1(B) specifies the type of operation.
R2(B) contains the file handle.
R3(D) contains data to be written.

If R2(B) is non-zero then OSARGS will do one of the following jobs on the file of which R2(B) is the handle:

```
R1(B) = 16_00 read sequential pointer to R3(D)
R1(B) = 16_01 writes R3(D) to sequential pointer
R1(B) = 16_02 read extent to R3(D)
R1(B) = 16_03 writes R3(D) to extent
R1(B) = 16_FF ensure this file is up to date on the media.
```

If the pointer is set to past the end of the file then the file will be padded such that intervening bytes read as nulls.

If R2(3) is zero then OSARGS will do one of the fallowing operations on the filing system:

```
R1(B) = 16_00 Return type of filing system in R1(D):
              0- No current filing system
              1- 1200 baud CFS
              2- 300 baud CFS
              3- ROM filing system
              4- Disc filing system
              5- Econet filing system
              6- Teletext/Prestel 'Telesoftware'

R1(B) = 16_01 Return address of rest of command line in I/O
              processor in R3(D).

R1(B) = 16_FF Ensure all open files (and any other necessary
              information) are up to date on the media.
```

- On Return:

F bit clear: Successful, R1(D) and R3(D) operation specific

F bit set: Error, R1(D) is the error code.

2.1.13 OS_BGET

- SVC code = 13
- Effect:

  Gets a byte from a specified OPEN file.

- On Call:

  R2(B) is the file handle

- On Return:

  F bit clear: implies a successfully completed transfer and R1(D) contains the byte read.

  F bit set: An error has been detected. If R1(D) = 16_1FE then an attempt has been made to read past end of file.

2.1.14 OS_BPUT

- SVC code = 14
- Effect:

  Puts a byte to specified OPEN file

- On Call:

  R2(B) is the file handle and R1(B) contains the byte to put.

- On Return:

  F bit clear: implies a successfully completed transfer
  F bit set: then error detected by MOST and R1(D) contains error number.

2.1.15 OS_GBPB

- SVC code =15

- Effect:

  Write/Read a group of bytes from a specified open file.

- On Call:

  R1(B) determines the type of operation:
  R1(B) = 16_01 Put byte using byte offset
  R1(B) = 16_02 Put byte ignoring byte offset
  R1(B) = 16_03 Get byte using byte offset

R1(B) = 16_04 Get byte ignoring byte offset
R1(B) = 16_05 Read title/cycle number/option/drive
R1(B) = 16_06 Read current directory
R1(B) = 16_07 Read current library
R1(B) = 16_08 Read file names

R2(B) is the file handle
R3(D) points to the start of the data area
R4(D) contains the number of bytes or items to transfer
R5(D) contains the optional value of the pointer

o On Return:

F bit set: implies error detected.

F bit clear: implies transfer completed successfully.
Values returned are as below:

R1(B) = 16_01 to 16_04:

R3(D) is updated (i.e. old data address plus the amount transferred)
R4(D) shows how much data has not been transferred
(and is usually zero)
R5(D) is the updated value of the pointer.

Pandora returns the value 16_1FE in R1 if an attempt is made to read past the end of file.

R1(B) = 16_05:

This returns in the area pointed at by the data pointer the title/cycle number/option and drive of the currently selected disk.
It is returned in this form:

(title length)(title)(option)(drive)

The cycle number, option and drive are single binary bytes.

R1(B) = 16_06:

This returns in the area pointed at by the data pointer the currently selected directory name.
It is returned in this form:

<length disk name><disk name>
<length directory name><directory name><priv>

<priv> = 16_00 =>Owner
<priv> = 16_FF =>Public

R1(B) = 16_07:

This returns in the area pointed at by the data pointer the currently selected library name.
It is returned in this form:

<length disk name><disk name>
<length library name><library name><priv>

<priv> = 16_00 =>Owner

<priv> = 16_FF =>Public

R1(B) = 16_08:

This returns the names of files in the current directory.
The format of the control block is similar to that for sequential files:

R2(B) returns the cycle number
R3(D) is the address to put the data
R4(D) is the file number of filenames to transfer
R5(D) is the file pointer

If the pointer is set to zero the search will begin with the first file. All registers are updated in a similar manner to the way pointers are updated for R1(B) = 16_01 to 16_04.
The format of the filenames is as follows:

<length filename 1><filename 1>
<length filename 2><filename 2>.....

2.2 Events and Event Control

There are two classes of events. One class corresponds to events being signalled by the I/O processor. Generally, these events are enabled/disabled using the relevant OS_BYTE call. Events in the other class do not correspond to OS_BYTE requests and are enabled using the SVC OS_CONTROL. For all events, the Kernel calls a routine to deal with it. By default the Kernel supplies a null event routine for each event. The call OS_HANDLER enables the control program to choose which routine should be called for each of the events.

Events enabled by OS_BYTE calls are as follows:

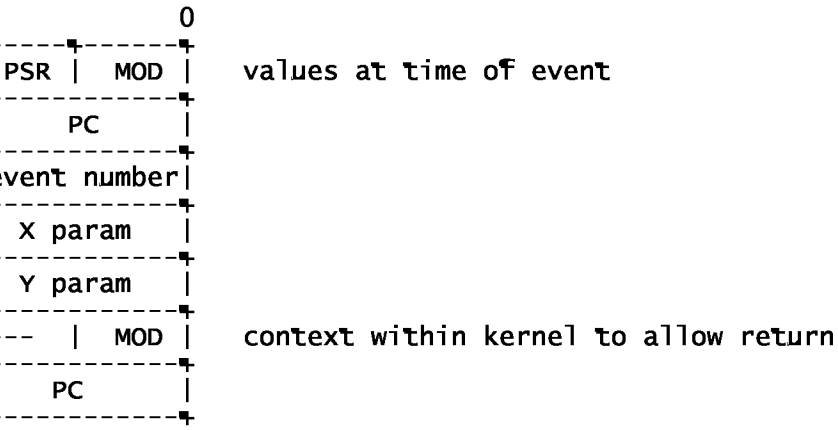
EVENT NUMBER	
0	Buffer Empty
1	Buffer Full
2	Keyboard Interrupt
3	ADC conversion complete
4	Start of TV field pulse
5	Interval timer crossing zero
6	Escape condition detected
7	RS423 error event
8	Network event
9	User event
255	Event unknown by Pandora

Events enabled by OS_HANDLER are:

EVENT NUMBER

251	Pandora SVC error detected
252	End of Pandora SVC call
253	Escape flag update
254	Unattributable error detected by I/O processor

For all events the event routine is entered:
(a) with bits I, S, U and T cleared in PSR,
(b) with R0-R7 containing what they did when the event occurred
(c) the supervisor stack looking as below:
NB. addresses increase upwards and leftwards.



If it is required to continue after the event then an RXP0 should be executed.

The event is described by the data addressed as 8(SP) through 16(SP):

For events 0-9:
8(SP) contains the "Y" parameter supplied by I/O processor
12(SP) contains the "X" parameter supplied by I/O processor
16(SP) contains the host event number (0-9)

For event 255:
8(SP) contains the "Y" parameter supplied by I/O processor
12(SP) contains the "X" parameter supplied by I/O processor

16(SP) contains the most event number (9-255)

For events 251 and 252:
8(SP) is undefined
12(SP) is undefined
16(SP) contains the event number (251 or 252)

For event 253:
8(SP) is undefined
12(SP) the value of the escape flag maintained by the MOST
e.g. =0 when escape flag clear and
=1 when escape flag set
16(SP) contains the F event number (i.e. 253)

For event 254:
8(SP) is the error number
12(SP) is undefined
16(SP) is the event number (254)

when enabled, type 251 events occur if a Pandora call ends in error (i.e. the F bit set).

If event 252 is enabled then when the next Pandora SVC returns (excluding return from OS_CONTROL and OS_HANDLER) an event is generated. Values of the stacked MOD, PSR and PC correspond to the instruction following the SVC instruction.

If both events 251 and 252 are enabled at the same time then event 252 takes preference - event 251 is ignored and the End_of_SVC event handler is entered. The F bit in the stacked PSR indicates whether the previous call was successful.

If it is wished to return to the place of interrupt then the routine should:

(1) Return after a short period
(2) Take care not to corrupt R0-R7.

Only calls to SVCs not involving TUBE protocols are allowed from event routines; SVCs in this category include those defined in sections 2.2, 2.3 and 2.4.

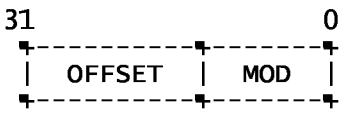
2.2.1 OS_HANDLER

- SVC code = 30
- Effect:

  Defines the supervisor routine that will be executed on a specific I/O processor event.

- On Call:

  R1(B) contains the event number
  R2(D) contains offset/module of routine entry in the standard 32016 external procedure descriptor format:



- On Return:

  F bit clear: successful and R2(D) contains offset/module of old handler
  F bit set : illegal event number
  All registers are as on entry

2.2.2 OS_CONTROL

- SVC code = 57
- Effect:

  Enables/disables events 251-254 (see section 2.2).
  (NB. other events are enabled/disabled using OS_BYTE)

- On Call:

  R1(B) is the event number
  R2(D) bit 0 =0, Disable event
  bit 0 =1, Enable event

- On Return:

  F bit clear : successful and R2(D) bit 0 contains the old status.
  F bit set : illegal event number.

2.3 The Virtual Dispatch Table

In order to service tube interrupts correctly and ensure that exceptions occurring whilst in the code of Pandora may be attributed to the originating SVC request, it is required that Pandora takes over the hardware Dispatch Table and presents to the control program a "virtual" table as a replacement. This table is called the Virtual Dispatch Table (VDT).

The control program informs Pandora about the whereabouts of the VDT using the "OS_SETVDT" call which sets the virtual dispatch table to a supervisor mode address supplied by the control program.

The user sees the VDT as a replica of the hardware defined Dispatch Table. When Pandora needs to inform the control program of an exception then it uses the VDT to obtain the external procedure descriptor of the service routine to call. Pandora enters the service routine, faking the entry so that it appears to the control program that the VDT is the true hardware defined dispatch table. After servicing the exception the return should be made using RETT or RETI where applicable.

- Traps occurring whilst in the control program are routed directly to the relevant routine supplied by the control program.
- NVIs and NMIs are passed on after the Kernel has filtered out any interrupts relevant to the workings of the TUBE.
- SVC exceptions are also filtered and any exceptions which are not known to Pandora are passed onto the service routine of the control program.

It must be noted that the control program should not alter the position of the hardware dispatch table. Alteration of INT_BASE will result in total Kernel malfunction. If the control program does not inform Pandora of the presence of a VDT then the Kernel indirects via the Kernel's default VDT to a Kernel supplied routine which prints out a diagnostic message on the VDU and enters a dynamic halt.

2.3.1 OS_SETVDT

- SVC code = 40
- Effect:

  Informs Pandora where the Virtual Dispatch Table resides in the supervisor space. The Kernel uses this table to indirect after a hardware exception to the application routine. OS_SETVDT is called from supervisor mode and the address supplied is, of course, taken to be within supervisor space. If OS_SETVDT is called from user space then Pandora passes the SVC on via the VDT.

- On Call:

  R1(D) contains address of VDT

- On Return:

  F bit clear always.
  All registers unchanged.

2.4 Miscellaneous

2.4.1 OS_ENTRY

- SVC code = 51
- Effect:

  Defines entry address in control program on subsequent <break>.

- On Call:

  R1(D) is the address of a control program which is entered on any subsequent break. On entry to the control program after a break R1 = 1.

- On Return:

  F bit clear always. All registers unchanged.

2.4.2 OS_VERSION

- SVC code = 52
- Effect:

  Returns the version number of Pandora.

- On Call:

  No parameters

- On Return:

  The version number is returned in R1(D). For example, version 3.23 returns R1(D) = #X00000323.

F bit is clear always.

2.4.3 OS_SETVDU

- SVC code = 53
- Effect:

  Code is loaded into the I/O processor to control the operation of its vdu driver.

- On Call:

  R1(D) <= 255

  R1(D) is the identity of a standard VDU handler:

  R1(D) = 0 The raw BBC MOS vdu handler (see BBC micro user guide)

  R1(D) = 1 The modified BBC MOS vdu handler used by mast languages on the 32016. Differences from the standard one are:

  #X0A : Newline code - generates a <CR> and then an <LF>
  #X0D : Carriage return - only does a <CF> never a <LF>
  #X1B : Down-line - moves vertically down in current column

OS_ASCI is identical to OS_WRCH when this handler is selected

R1(D) = 2 -> 255 reserved

R1(D) > 255

R1(D) contains the address of a vdu handler data structure. This contains 6502 code, relocation bit map, newline string and other information in the format in Appendix D.

o On exit:

F bit clear: successful
R1(D) = The identity of the previous handler
(so you can restore it)

F bit set: error, the handler was not inserted because:
R1(D) = -1 The format of the data structure was incorrect.
R1(D) = -2 A bad newline string was supplied
R1(D) = -3 Incompatible tube software
R1(D) = -4 Insufficient space in the host

2.4.4 OS_CONFIG

o SVC code = 54

o Effect:

Returns the value of the second processor configuration switches.

o On Call:

No parameters

o On Return:

F bit clear always, R1(D) contains the configuration information:

Bit in R1	Meaning
0	=1, FPU present =0. FPU absent
1	=1, MMU present =0, MMD absent
2-7	RESERVED
8-31	=0 always

2.4.5 OS_PRIV

- SVC code = 55
- Effect:

  Controls which SVC operations Pandora will carry out.

- On Call:

  R1(B) = SVC number.

  R2 bit 0 =0 -> pass this SVC on via the VDT
  =1 -> carry out this SVC in Pandora

  All other bits reserved

- On Return:

  F bit clear, operation successful F1(D)
  previous status of this SVC in bit 0

  F bit set, error R1(D) contains the error code
  =16_101 Attempt to modify status of non-Pandora SVC
  =16_102 Attempt to modify the status of OS_PRIV
  =16_103 Called from user mode

2.4.6 OS_NOLOAD

- SVC code = 56
- Effect:

  Controls whether or not OS_CLI calls are allowed to cause LOADS into the second processor store. After this call any OS_CLI operation which attempts to load data into the 32016 store will fail.

- On Call:

  R1(D) bit 0 = 0, Disallow LOADs from OS_CLI operations.
  bit 0 = 1, Allow LOADs from OS_CLI operations.
  All other bits reserved.

- On Return:

  R1(D) bit 0 contains the old status.

2.4.7 OS_ERROR

- SVC code = 31
- Effect:

  Reads Pandora error message. After reading the length of the current message is set to zero.

  N.B. In order to avoid the current message being overwritten the OS_ERROR should be called prior to any further Kernel calls.

- On Call:

  R1(D) points at error message buffer of requestor

  R2(D) is maximum length of error message to be accepted

- On Return:

  F bit = 0 always.

  R2(D) is length of message.
  (truncated to the maximum length of supplied buffer)

  R3(D) is the error number

  All other registers are unchanged.

2.4.8 OS_EXIT

- SVC code = 17
- Effect

  Causes control to be passed back to the control program. The entry point at which the control program is entered is defined using OS_ENTRY. On entry to the control program R1=2 indicating that entry was caused by OS_EXIT.

- On Call:

  No parameters

- On Return:

  This call does not return.

2.4.9 OS_ESCAPE

- SVC code = 20
- Effect:

  Reads the kernel escape flag.

- On Call:

  No parameters

- On Return:

  F bit clear: no escape detected
  F bit set: escape has been detected
  All other registers are unchanged

2.4.10 OS_SVR

- SVC code = 32

- Effect:

  Clear the U bit in the PSR register.
  All other PSR are unchanged.

- On Call:

  No parameter

- On return

  F bit = 0 always
  All other register unchanged

2.4.11 OS_INDIR

- SVC code = 0
- Effect:

  Calls the SVC request passed in R0.

- On Call:

  Registers contain parameters appropriate for the called SVC.

o On Return:

Registers and flags contain return values from the called SVC.

3. Entry to the Control Program

o Initial Entry

After reset (power on, <ctrl-break> or <shift-break>), Pandora checks whether the HOST has sent a NS32016 program (i.e. ROM/FILE header has required type -32016- and copyright message - see appendix C) down the TUBE. If so, then the program is considered the control program and entered. Otherwise, Pandora enters the default control program - the command interpreter of PANDORA. The default control program is a simple program which reads in command lines and passes them (using OS_CLI) to the BBC MOS. This facilitates the loading and running of user programs.

Note that Pandora enters the control program with no stacked information and takes no advantage of the position of the supervisor stack. Therefore the control program may reposition the supervisor stack to any convenient area of memory as long as there is always room for kernel to stack 16_400 bytes.

o Re-entry

The control program can be re-entered by
(a) <break>
(b) <shift-break>
(c) under program control via OS_EXIT
(d) via OS_CLI (i.e. load and run a file with a valid 32016 header)

o On entry R1 indicates how the control program was entered:

R1=0 initial entry (power-on or <ctrl-break>)
R1=1 entry after any variety of <break>
R1=2 entry via OS_EXIT
R1=3 entry via OS_CLI

On entry U = 0, S = 0 and I = 1 in the PSR register.

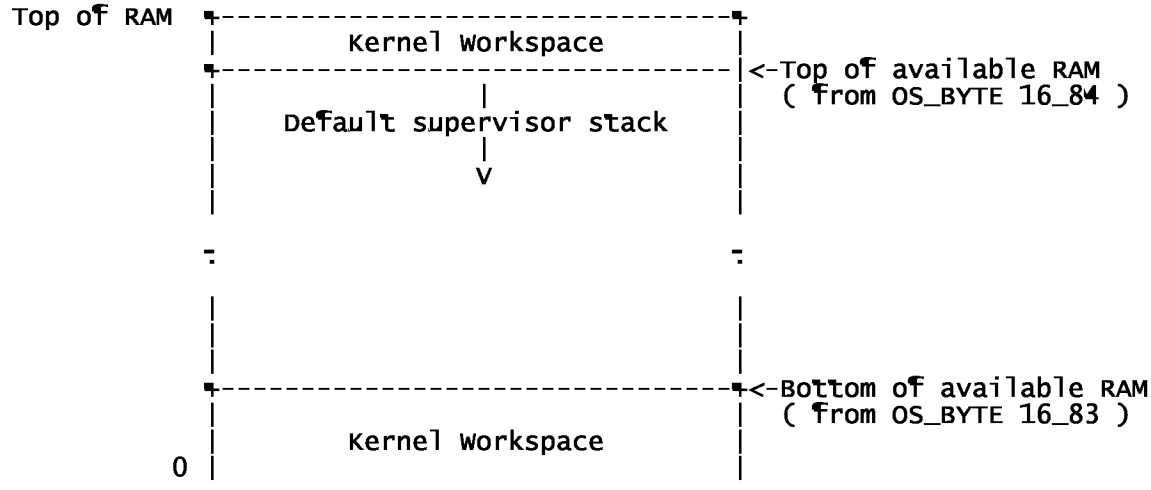
All data structures in the Pandora workspace are re-initialised:
the VDT reverts back to the default table,
events are disabled,
event routines revert to the default (null) routines,
the table determining which SVCs are valid is reset.

The layout of memory on entry to the control program is described the next section.

4. Memory Map

Pandora runs from ROM but requires two areas of RAM for workspace, one at the bottom and one at the top of memory. The limits of the memory area available to the application system can be discovered by the use of OS_BYTE calls 16_83 and 16_84.

Store layout



Appendix A - Summary of SVC Calls

Name	SVC Number Dec	Hex	Description
			MOS Request
OS_WRCH	1	&01	Write character to output stream(s)
OS_STRING	2	&02	Write string to output stream(s)
OS_ASCI	3	&03	Write char with CR-> LF/CR to output
OS_NEWL	4	&04	Write LF/CR to output stream(s)
OS_RDCH	5	&05	Read character from input stream
OS_BYTE	6	&06	A large group of miscellaneous calls
OS_WORD	7	&07	A large group of miscellaneous calls
OS_CLI	8	&08	Pass command line to BBC to execute
OS_FILE	10	&0A	Manipulate file data and catalogue
OS_FIND	11	&0B	Open file for read/write/update
OS_ARGS	12	&0C	Read/write an open files attributes
OS_BGET	13	&0D	Get byte from file
OS_BPUT	14	&0E	Put byte to a file
OS_GBPB	15	&0F	Read/Write bytes to a file
OS_INLINE	21	&15	Output inline string
			Control of events
OS_HANDLER	30	&1E	Set routine to be called on event
OS_SVR	32	&20	Enter supervisor mode
OS_CONTROL	57	&39	Enable/Disable events 251-255
			Virtual Dispatch Table Management
OS_SETVDT	40	&28	Positions the Virtual Dispatch Table
			Miscellaneous
OS_INDIR	0	&00	Request SVC call passed R0
OS_EXIT	17	&11	Returns control to control program
OS_DEBUG	18	&12	

OS_ESCAPE	20	&14	Read value of kernel escape flag
OS_ERROR	31	&1F	Read error message from I/O processor
OS_ABORT	33	&21	Jump through ABORT vector
OS_SPLIT	50	&32	
OS_ENTRY	51	&33	Defines <break> entry address
OS_VERSION	52	&34	Returns the version number of kernel
OS_SETVDU	53	&35	Install vdu handler in the I/O processor
OS_CONFIG	54	&36	Returns the value of the H/W switches
OS_PRIV	55	&37	Control passing of SVC operations
OS_NOLOAD	56	&38	Allow/disallow LOAD from within OS_CLI

Numerical List of SVC calls

SVC Number Dec	Hex	Name	Description
0	&00	OS_INDIR	Request SVC call passed R0
1	&01	OS_WRCH	Write character to output stream(s)
2	&02	OS_STRING	Write string to output stream(s)
3	&03	OS_ASCI	Write char with CR-> LF/CR to output
4	&04	OS_NEWL	Write LF/CR to output stream(s)
5	&05	OS_RDCH	Read character from input stream
6	&06	OS_BYTE	A large group of miscellaneous calls
7	&07	OS_WORD	A large group of miscellaneous calls
8	&08	OS_CLI	Pass command line to BBC to execute
10	&0A	OS_FILE	Manipulate file data and catalogue
11	&0B	OS_FIND	Open file for read/write/update
12	&0C	OS_ARGS	Read/write an open files attributes
13	&0D	OS_BGET	Get byte from file
14	&0E	OS_BPUT	Put byte to a file
15	&0F	OS_GBPB	Read/Write bytes to a file
17	&11	OS_EXIT	Returns control to control program
18	&12	OS_DEBUG	
20	&14	OS_ESCAPE	Read value of kernel escape flag
21	&15	OS_INLINE	Output inline string
30	&1E	OS_HANDLER	Set routine to be called on event
31	&1F	OS_ERROR	Read error message from I/O processor
32	&20	OS_SVR	Enter processor supervisor mode
33	&21	OS_ABORT	Jump through ABORT vector
40	&28	OS_SETVDT	Positions the Virtual Dispatch Table
50	&32	OS_SPLIT	
51	&33	OS_ENTRY	Defines <break> entry address
52	&34	OS_VERSION	Returns the version number of kernel
53	&35	OS_SETVDU	Install vdu handler in the I/O processor

54	&36	OS_CONFIG	Returns the value of the H/W switches
55	&37	OS_PRIV	Control passing of SVC operations
56	&38	OS_NOLOAD	Allow/disallow LOAD from within OS_CLI
57	&39	OS_CONTROL	Enable/disable events 251-255

Appendix B - OSWORD

Table of OSWORD calls

R1(B)	DESCRIPTION
00	Read a line from currently selected input.
01	Read system time.
02	Write system time.
03	Read system time interval counter.
04	Write to system time interval counter.
05	Read I/O processor.
06	Write to I/O processor.
07	Make a sound.
08	Define an envelope.
09	Read pixel value.
0A	Read_character definition for given character.
0B	Read palette value for given logical colour.
0C	Write palette value for given logical colour.
0D	Read last two graphics cursors.
0E	Read real-time clock
0F	Write to real-time clock
7D	Get DFS cycle number.
7E	Read current DFS disc size.
7F	Bash disc controller.

OSWORD calls - descriptions

OSWORD call 16_00

- Read a line from the currently selected input. R2(D) points to a buffer containing:

  0(R2) The address of the buffer for the input line
  4(R2) The length of the buffer
  5(R2) Input character lower bound
  6(R2) Input character upper bound

  Characters will only be entered in the buffer if they are within the range specified by locations 5(R2) (inclusive lower bound) and 6(R2) (inclusive upper bound). During input, DEL deletes the last character input and NAK (ctrl/u) deletes the entire input line.

- On Exit:

F = 0 Indicates that a carriage-return (ASCII 16_0D) terminated the line.

F = 1 Indicates that an escape condition occurred.

R3D is set to the length of the input line (including the CR if F = 0).

OSWORD call 16-01

o Read system time

The five byte system time is read into locations 0(R2) (lsb) to 4(R2) (msb).

System time is set to zero by a hard reset.

OSWORD call 16_02

o Write system time.

The five byte system time is reinitialised with the value at locations 0(R2) (lsb) to 4(R2) (msb).

OSWORD call 16_03

o Read time interval counter.

The system time interval counter is a five byte incrementing counter. It is reset to zero by any reset. When the system time interval counter crosses zero an event is generated. The five byte interval counter is read into locations 0(R2) (lsb) to 4(R2) (msb).

OSWORD call 16_04

o Writes to the system time interval counter.

The five interval counter is initialised with the value at locations 0(R2) (lsb) to 4(R2) (msb). (Note that the counter is an incrementing counter, thus a value of 16_FFFFFFFFFF would give a time interval of 1 centisecond).

OSWORD call 16_05

o Read a byte from I/O processor.

Uses a 32 bit address taken from locations 0(R2) to 3(R2), result returned in 4(R2).

Useful when used in conjunction with the tube - the tube will only pass the byte if the address is in the top range.

OSWORD call 16_06

o Write a byte to I/O processor.

Uses a 32 bit address taken from locations 0(R2) to 3(R2). Byte for writing is taken from 4(R2).

OSWORD call 16_07

o Make a sound.

The 8 bytes at locations 0(R2) to 7(R2) are treated as 4 2-byte values, 0(R2) lsb, 1(R2) msb etc. These four values define the sound effect. See the sound specification in the new user guide for more details.

OSWORD call 16_08

o Define an envelope.

The 13 bytes at locations 1(R2) to 13(R2) are used to define the envelope indicated by 0(R2). See the sound specification in the new user guide for more details.

OSWORD call 16_09

o Read pixel value.

The 4 bytes at locations 0(R2) are 3(R2) are treated as 2 2-byte values. 0(R2) (lsb) and 1(R2) (msb) represent an X-coordinate, 2(R2) (lsb) and 3(R2) (msb) represent a Y-coordinate. The value of the pixel at the addressed coordinate is deposited in location 4(R2). An invalid coordinate returns a value of 16_FF.

OSWORD call 16_0A

o Read character definition far a given character.

The character definition for the character given in location 0(R2) is deposited in locations 1(R2) (top row) to 8(R2) (bottom row).

OSWORD call 16_0B

o Read the palette value for a given logical colour.

The 4-byte physical colour definition for the logical colour given at location 0(R2) is deposited at locations 1(R2) to 4(R2).

OSWORD call 16_0C

o Write the palette value for a given logical colour.

The 4-byte physical colour definition at locations 1(R2) to 4(R2) is assigned to the logical colour 0(R2).

OSWORD call 16_0D

o Read graphics cursor positions.

The last two positions visited by the graphics cursor are returned as X, Y, X, Y (each two byte integers).

OSWORD call 16_0E

o Read the real-time clock.

OSWORD call 16_0F

o Write to the real-time clock.

OSWORD call 16_7D

o Read the DFS cycle number (of currently selected directory). The cycle number is returned in the location (R2 D).

OSWORD call 16_7E

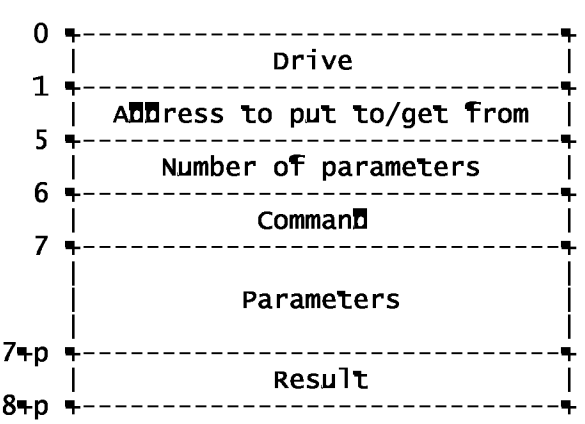
o Read the DFS disc size (in bytes of the currently selected disc).

The four byte disc size is returned in locations 0(R2) to 3(R2).

OSWORD call 16_7F

o Access the disc controller.

(R2 D) points at this control block:



The drive byte will select drives 0-3 in the standard DFS manner (i.e. drive 2 is side two of drive 0). If set to 16_FF however no drive will be

selected. It should be noted that no command will work if the drive is not spinning. The address bytes are where data should be taken from/put to depending on the type of command. If no data transfer takes place, then they may be ignored.

(R2 D) + 5 determines the number of parameters for the disc controller

after the command byte.

The command byte is the command for the disc controller command register. Bit 6 should be set and bit 7 should be unset as the drive selection is done automatically and these bits are used for it. Thus the command for read sector becomes 16_53 and write sector 16_4B. The parameter bytes are sent to the disc controller parameter register after the command byte is sent to the command register.

The result returned at the end of the operation is read from the result register and stored in the next byte after the parameters.

Note that the direction of any possible data transfer is determined by bit 4 of the command byte: a 1 means that you are reading from the disc, a 0 means that you are writing to it.

Appendix C - Header of 32016 Control Programs

The fragment of assembly code below describes the format of the header which enables the code to be entered as a control program on the 32016 second processor.

```
; ROM format constants

ROM_SERVICE          EQU     #b10000000
ROM_LANGUAGE         EQU     #b01000000
ROM_TUBE             EQU     #b00100000
ROM_ELK              EQU     #b00010000
ROM_PROC_MASK        EQU     #x0D
ROM_6502             EQU     #x00
ROM_TURBO_6502       EQU     #x01
```

```
ROM_PROC_4              EQU     #x04
ROM_PROC_5              EQU     #x05
ROM_Z80                 EQU     #x08
ROM_32016               EQU     #x09
ROM_286                 EQU     #x0C
ROM_ARM                 EQU     #x0D

I_6502_JMP              EQU     #x4C
I_6502_RTS              EQU     #x60
ROM_MOST_ADDRESS        EQU     #x8000

ROM_ADDRESS             EQU     #x00000200
ROM_TYPE_BITS           EQU     ROM_LANGUAGE | ROM_TUBE | ROM_PROC_32016


                        RELORG  ROM_ADDRESS

ROM_BASE                DCB     I_6502_JMP
                        DCW     LANGUAGE_6502-ROM_BASE+ROM_MOST_ADDRESS
                        DCB     I_6502_JMP
                        DCW     SERVICE_6502-ROM_BASE+ROM-MOST_ADDRESS
ROM_TYPE                DCB     ROM_TYPE_BITS
ROM_COPYRIGHT_OFFSET    DCB     ROM_COPYRIGHT - ROM_BASE
ROM_VERSION_NUMBER      DCB     2
ROM_NAME                DCS     'BCPL-32016', 0
ROM_VERSION_STRING      DCS     'Version 0.35 of 10th August 1983'
;
ROM_COPYRIGHT           DCS     0, '(C) Acorn 1983', 0
ROM_RELOCATION          DCD     ROM_ADDRESS
ROM_EXECUTION           DCD     BOOT-ROM_BASE


LANGUAGE 6502           DCB     #x58,#xA9,#xAA,#xA2,#x00,#xA0,#xFF,#x20
                        DCB     #xF4,#xFF,#x86,#x00,#xA9,#xAB,#xA2,#x00
                        DCB     #xA0,#xFF,#x20,#xF4,#xFF,#x86,#x01,#xA9
                        DCB     #xFC,#xA2,#x00,#xA0,#xFF,#x20,#xF4,#xFF
                        DCB     #x8A,#xA8,#x88,#x30,#x1B,#xB1,#x00,#x29
                        DCB     #x4D,#xC9,#x40,#xF0,#x54,#xD0,#xF3,#xA9
                        DCB     #x7E,#x20,#xF4,#xFF,#x20,#xE7,#xFF,#x00
                        DCB     #x99,#x43,#x73,#x63,#x61,#x70,#x65,#x00
                        DCB     #x18,#x20,#xE7,#xFF,#xAD,#x01,#x80,#x69
                        DCB     #x8A,#x8D,#x02,#x02,#xAD,#x02,#x80,#x69
```



```
                        DCB     #x00,#x8D,#x03,#x02,#xA9,#x2A,#x20,#xEE
                        DCB     #xFF,#xA9,#x06,#x85,#x00,#xA9,#x00,#x85
                        DCB     #x01,#xA9,#xF0,#x85,#x02,#xA9,#x00,#x85
                        DCB     #x03,#xA9,#x7E,#x85,#x04,#xA9,#x00.#xAA
                        DCB     #xA8,#x20,#xF1,#xFF,#xB0,#xB9,#xA2,#x06
                        DCB     #xA0,#x00,#x20,#xF7,#xFF,#xD0,#xD5,#xF0
                        DCB     #xD3,#x98,#xAA,#xA9,#x8E,#xA0,#xFF,#x4C
                        DCB     #xF4,#xFF,#x20,#xE7,#xFF,#xA0,#x01,#xB1
                        DCB     #xFD,#xF0,#xAD,#x20,#xEE,#xFF,#xC8,#xD0
                        DCB     #xF6

SERVICE_6502
; If the ROM contains OS services, then set the ROM_SERVICE bit in
; the ROM_TYPE byte, and insert the 6302 service code here.

                        DCB     I_6502_RTS
```

Appendix D - VDU handler data structure

Section missing.