CS502 System Calls

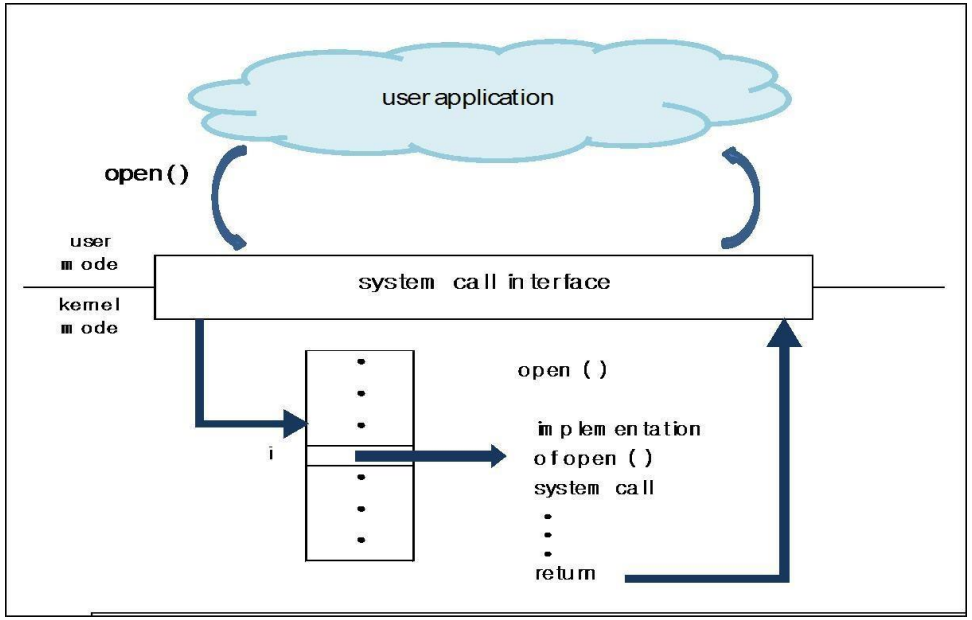
July, 2018

Rev 4.50

Introduction

This document contains a description of all the system calls that may be supported by the CS502 Operating System. The document has the following components:

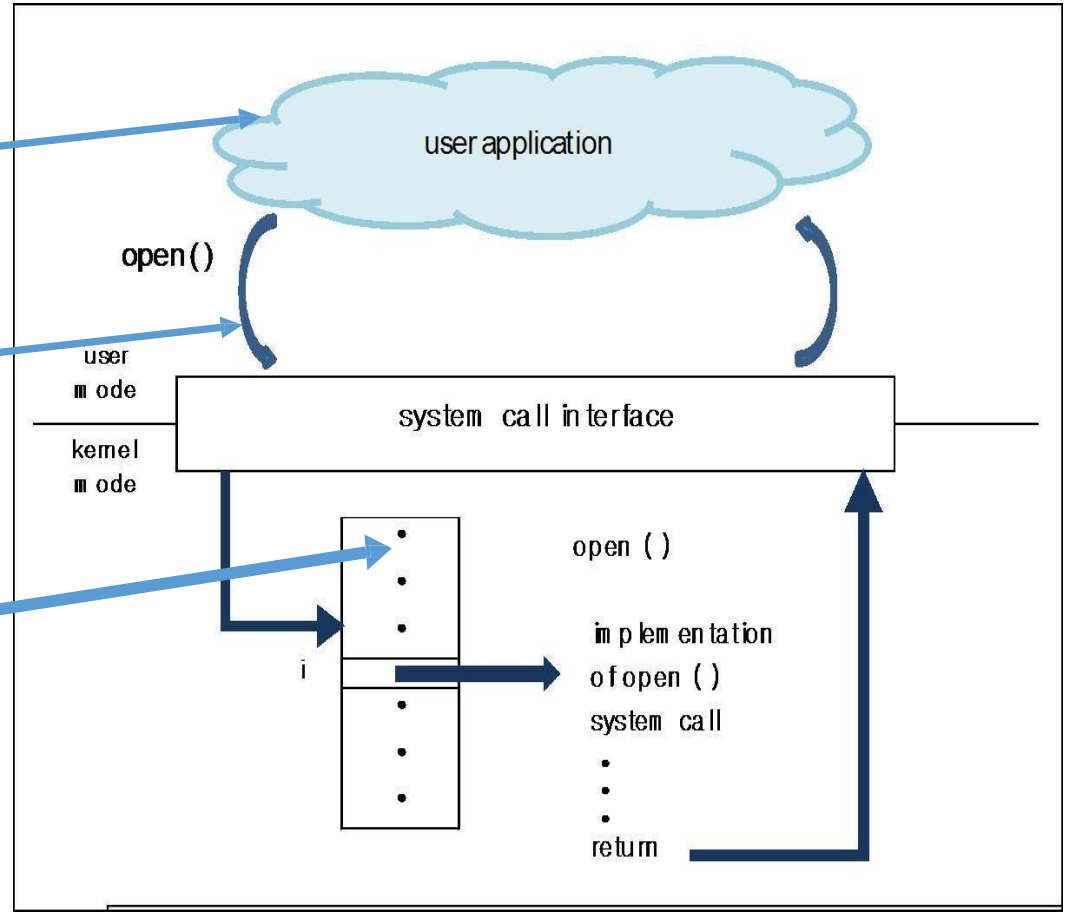
- This introduction – an overview and where to find other information about system calls.
- System Calls that don't involve the File System.
- A description of the File System. That description is appropriate here because a file system is a property of an Operating System and is NOT defined by the underlying hardware.
- System Calls that DO involve the File System.



Introduction

With reference to this image:

- The user application in our case is the set of tests in test.c.
- The System Call interface is defined in syscalls.h. Here you'll find a set of macros that encapsulate the workings of the call. Here also you'll find a unique identifier for each System Call Number.
- All the action required by a system call takes place in the OS. The code for this is set up at svc() in base.c. That's where you do all your work.



System Calls GET_TIME_OF_DAY

Usage of GET_TIME_OF_DAY

long CurrentTime;

GET_TIME_OF_DAY (&CurrentTime);

Causes the Operating System to ask for the time as seen by the Z502 hardware. This time is arbitrary and dependent on the simulation. There is no relation between the CurrentTime used here and a physical clock – either one represented by Linux or Windows OS or by the clock on the wall. This call is guaranteed to return a unique time; two “simultaneous” calls to the hardware clock will return different times. This guarantee is how Operating Systems are able to determine uniqueness and which action should proceed ahead of others.

There is no error possible with this call.

System Calls SLEEP

Usage of SLEEP

long TimeToSleep;

SLEEP (TimeToSleep);

Causes the Operating System to invoke the hardware timer. If TimeToSleep is zero or negative, the Operating System should return immediately. Otherwise, the OS should start a hardware timer and suspend this process. When the timer, at some later time, causes an interrupt, the OS will again schedule this process to continue its execution. Note that the time elapsed during the sleep will almost certainly be longer than TimeToSleep.

There is no error possible with this call. Control returns to this process when the time interval has completed. OR returns immediately if the caller has given a negative or zero time.

System Calls CREATE_PROCESS

Usage of CREATE_PROCESS

```
char ProcessName[N];
void *StartingAddress;
long  InitialPriority;
long  ProcessID;
long  ErrorReturned;
```

CREATE_PROCESS(ProcessName, StartingAddress, InitialPriority, &ProcessID, &ErrorReturned);

Causes the Operating System to create a process. This process will have name "ProcessName ", will begin execution at location/address " StartingAddress " (in essence, this is the name of a routine), and at the start of execution has a priority of InitialPriority. The system call returns the ProcessID of the newly created process. If an error occurs with this call, then ErrorReturned is not equal to ERR_SUCCESS and the call is not completed.

Possible errors include:
Illegal or duplicate process names
If the starting address is illegal, the code won't compile.
Illegal priority
Others

System Calls GET_PROCESS_ID

Usage GET_PROCESS_ID

char ProcessName[N];
long ProcessID;
Long ErrorReturned;

GET_PROCESS_ID(ProcessName, &ProcessID, &ErrorReturned);

Causes the Operating System to find the identity of a process having a name ProcessName. If such a process exists, the ProcessID of that process is returned. If the process does not exist, then ProcessID is not modified and an error of ErrorReturned not equal to ERR_SUCCESS is returned. It is up to the OS to determine what that error will be. A number of the tests use this feature to ensure that child processes have terminated.

If the ProcessName in this call is a null string (“”), then it is assumed that the calling process is asking for the ID of its own process.

Possible errors include:

- Process does not exist.
- Illegal ProcessName

System Calls TERMINATE_PROCESS

Usage TERMINATE_PROCESS

```
 long ProcessID;
long  ErrorReturned;
```

TERMINATE_PROCESS(ProcessID, &ErrorReturned);

Terminate the process whose PID is given by " ProcessID ". This is limited to processes that are children of the process doing the TERMINATE.
If ProcessID = -1, then terminate self.
If ProcessID = -2, then terminate self AND any children, grandchildren, etc. of the process calling the TERMINATE. This will not necessarily end the simulation.
An attempt to terminate a non-existent process results in an error being returned. Upon termination, all information about a process is lost and it will never run again. An error is returned if a process with that PID doesn't exist.

Possible errors include:

- Process does not exist.
- The target process isn't in the hierarchy (isn't a child of) the requester.

System Calls SUSPEND_PROCESS

Usage SUSPEND_PROCESS
long ProcessID;
long ErrorReturned;

SUSPEND_PROCESS(ProcessID, &ErrorReturned);

Suspend the process whose PID is given by " ProcessID ". If ProcessID = -1, then suspend self – but be careful that there is an entity that will wake you up in the future. Suspension of an already suspended process results in no action being taken. Upon suspension, a process is removed from the ready queue; if a process is on the Timer Queue or a Disk Queue at the time of this call, it stays there until the action it was waiting for is completed; then it is NOT put on the ready queue. The process will not run again until that process is the target of a RESUME_PROCESS. An error is returned if a process with the requested PID doesn't exist.

Possible errors include:
Process does not exist.

System Calls

RESUME_PROCESS

Usage RESUME_PROCESS

```
long  ProcessID;
long  ErrorReturned;
```

RESUME_PROCESS(ProcessID, &ErrorReturned);

Resume the process whose PID is given by " ProcessID ". Resumption of a non-suspended process results in an error. Upon resuming, a process is placed on the ready queue. An error is returned if a process with the requested PID doesn't exist.

Possible errors include:

- Process does not exist.
- Process is not suspended.

System Calls CHANGE_PRIORITY

Usage CHANGE_PRIORITY

```
long  ProcessID;
long  NewPriority;
long  ErrorReturned;
```

CHANGE_PRIORITY(ProcessID, NewPriority, &ErrorReturned);

Change the priority of the process whose PID is given by " ProcessID ". If ProcessID = -1, then change self. The result of a change priority takes effect immediately.

Possible errors include:

- Process does not exist.
- NewPriority is an illegal number.

System Calls SEND_MESSAGE

Usage SEND_MESSAGE

```
long   TargetProcessID;
char MessageBuffer[N];
long  MessageSendLength;
long  ErrorReturned;
```

SEND_MESSAGE(ProcessID, MessageBuffer, MessageSendLength, &ErrorReturned);

Send a message to the target process. When the target does a "RECEIVE_MESSAGE", place data from this send in the message buffer of that target. MessageSendLength is the size of the MessageBuffer; MessageSendLength must be equal in size to the string of data that is actually sent; so this parameter is a buffer size rather than a message size – it ensures that the receiver has enough space to receive the sent number of bytes. If the target ProcessID = -1, then broadcast the message to all potential receivers (although this message will actually be intercepted by only one of those receivers).

Possible errors include:

- Process does not exist.
- MessageSendLength less than or equal 0.

System Calls RECEIVE_MESSAGE

Usage RECEIVE_MESSAGE

```
long  SourcePID;
char MessageBuffer[N];
long  MessageReceiveLength;
long  MessageSendLength;
long  MessageSenderPid;
long  ErrorReturned;
```

RECEIVE_MESSAGE(SourcePID, MessageBuffer, MessageReceiveLength , &MessageSendLength, &MessageSenderPid , &ErrorReturned);

Receive a message from SourcePID. So SourcePID says what processes we will receive from. If SourcePID = -1, then receive from any sender who has specifically targeted you or who has done a broadcast. If -1 is used, then MessageSenderPid contains the PID of the process that did the send. The Operating System will place that message in message buffer if it is less than MessageReceiveLength bytes long. MessageReceiveLength is the size available in the receive buffer. Return the size of the send buffer in MessageSendLength. In general, a RECEIVE_MESSAGE system call causes the receiver process to suspend itself until the SEND is made. Devise appropriate rules of behaviour for the sender and receiver.

Possible errors include:

A bad SourcePID means no message will ever be received.

System Calls PHYSICAL_DISK_READ

Usage PHYSICAL_DISK_READ
Long DiskID;
long Sector;
char ReadBuffer[PGSIZE];

PHYSICAL_DISK_READ(DiskID, Sector, ReadBuffer);

Read data from location from sector “Sector” on the disk with ID DiskID to the location ReadBuffer that has been allocated by the calling program.
Note that this is a very low level system call. In most Operating Systems, knowledge of the location of physical data is reserved for a file system. The call as implemented here assumes that the user program understands the physical disk.

Possible errors include:

NONE! If there is an error in one of these parameters, for instance, an illegal DiskID or Sector, this call has no way of capturing it. If those erroneous parameters are passed on to the disk, the disk itself will interrupt with an error and will not complete the action.

System Calls

PHYSICAL_DISK_WRITE

Usage PHYSICAL_DISK_WRITE
long DiskID;
long Sector;
char WriteBuffer[PGSIZE];

PHYSICAL_DISK_WRITE(DiskID, Sector, WriteBuffer);

Write data from location WriteBuffer to sector “Sector” on the disk with ID DiskID.
Note that this is a very low level system call. In most Operating Systems, knowledge of the location of physical data is reserved for a file system. The call as implemented here assumes that the user program understands the physical disk.

Possible errors include:

NONE! If there is an error in one of these parameters, for instance, an illegal DiskID or Sector, this call has no way of capturing it. If those erroneous parameters are passed on to the disk, the disk itself will interrupt with an error and will not complete the action.

System Calls DEFINE_SHARED_AREA

Usage DEFINE_SHARED_AREA

```
long  StartingAddress;
long  PagesInSharedArea;
char AreaTag[MAX_TAG_LENGTH];
long  NumberPreviousSharers;
long  ErrorReturned;
```

DEFINE_SHARED_AREA(StartingAddress, PagesInSharedArea, AreaTag, &NumberPreviousSharers, &ErrorReturned);

“StartingAddress” is the virtual memory address at which the shared area should begin.
“PagesInSharedArea” is the number of pages that are to be in the shared area. Shared areas must encompass entire pages - you can't split a page with some of it being shared and some of it unshared.
The AreaTag is a label used to enumerate which shared area is being asked for. Programs can DEFINE multiple shared areas each having a separate tag. Programs specifying different AreaTags will get unrelated shared areas. It would presumably be legal for a process to do two DEFINEs, each with a different starting address, but having the same AreaTag, such that the same physical memory would be mapped to two separate virtual locations in the same process.
The NumberPreviousSharers indicates the number of other processes that currently have this area_tag shared area DEFINED. Thus the Operating System would return a "0" for the first caller, a "1" for the next, and so on. This has the effect of giving a unique ID to each sharer.
Possible errors include:

Any of the parameters in this call could be illegal.

NON System Calls

MEM_READ, MEM_WRITE, and READ_MODIFY

These "calls" are implemented by user programs, but instead of calling the Operating System, they instead call the Z502 hardware directly. They are thus described in “Z502 Hardware” under the explanation of the hardware interface. There is no action required by the OS to directly handle these calls.

Z502 Disk Structure

There are many ways to lay out data on a disk. In this project, it's possible to let you, the student, develop your own style of structure and information on the disk. I've decided to instead give you a structure and methodology. There are three reasons I'd like to go in that direction.

1) Giving you a structure makes your job easier – you don't need to develop something from scratch.
2) What I'm giving you is a structure that represents what you might find on a real disk – you'll learn by understanding this format.
3) I've developed a DiskCheck() that will execute on the structure you have placed on a disk – this allows you to be assured that your disk is correct – I would not be able to build such a Method for you without having a set of rules on how a disk should look.

Z502 Disk Structure

Sector 0

Definition Of Sector 0: This is the first block on a disk. It is produced when you do a software Format of the disk. It defines various segments of the disk necessary for successful operation.

Byte Offset	Bytes in Field	Field	Description
0	1	Disk ID	The number you have given to the disk. It will be easiest if you assign the same number as the hardware uses, but it's your choice.
1	1	Bitmap Size	How large is the bitmap. You can determine your own size. The size will be 4 X BitmapSize So if this field contains a 3, then 4 X 3 = 12 and 12 blocks will be assigned to the bitmap.
2	1	RootDir Size	How large is the Root Directory. You can determine your own size. The size will be RootDirSize So if this field contains a 2, then 2 blocks will be assigned to the Root Directory. Note that the size of a directory is determine by the number of subdirectories and files that are in that directory and the size does NOT include any index blocks that are associated with those subdirectories and files.
3	1	Swap Size	How large is the Swap Space. You can determine your own size. The size will be 4 X SwapSize So if this field contains a 2, then 4 X 2 = 8 and 8 blocks will be assigned to the Root Directory.
4	2	Disk Length	This is the number of sectors you are using on the disk. Note Byte 5 is Most Significant Byte, Byte 4 is Least Significant Byte
6	2	Bitmap Location	The starting sector number where the bitmap is located. Note Byte 7 is MSB, Byte 6 is LSB
8	2	RootDir Location	The starting sector number where the Root Directory is located. Note Byte 9 is MSB, Byte 8 is LSB
10	2	Swap Location	The starting sector number where the Swap Space is located. Note Byte 11 is MSB, Byte 10 is LSB
12	4	RESERVED	For this release, these bytes must be set to '\0'.

Z502 Disk Structure

Bitmap

The function of the bitmap is as follows: There is one bit assigned for each sector on the disk. If that bit is set, then the sector is in use; if that bit is clear, then that sector is unused. The bitmap requires a number of sectors on the disk for its own storage.

So IF you had a bitmap showing ONLY Sector 0 in use, it would be represented by “10000000,00000000,00000000, . . .”, and if you were to print out that bitmap, it would look like 80 00 00 00 . . .

Z502 Disk Structure

Header For Disk or File

Every File or Directory has a Header Block. This contains the essential information for that entity. Among the items associated with this Block is a pointer to an index. This index points to the DATA for this file or to the Files and Directories within a directory. Details about these fields are given in the following slides.

Byte Offset	Bytes in Field	Field	Description
0	1	Inode	A unique ID for this file or directory. This Inode is unique across all disks on the entire system. This means there are NOT TWO identical Inodes on the Z502.
1	7	Name	The ASCII text representing the name. Only letters and numbers are allowed in names. Note that this can be 7 characters so MAY not have a delimiter. The field is left justified so that Byte Offset 1 contains the first character of the name. Unused characters in the name contain "\0".
8	1	File Description	See the format of the Description Field below
9	3	Creation Time	Contains the time at which the file/directory was created. This is the time of creation as found by asking the hardware for the time. Byte 11 is the MSB and Byte 9 is the LSB.
12	2	Index Location	The sector number where the Index Sector containing the sector numbers for this file is located. Note Byte 13 is MSB, Byte 12 is LSB
14	2	File Size	The number of bytes occupied by the data portion of this file. It's necessary to allocate sectors to hold the entire data portion of the file, but the last sector need not be totally filled. Byte 15 is MSB, Byte 14 is LSB

Z502 Disk Structure

Header For Disk or File

Further explanation of the Fields in a Header:

File Description Field

Bits	Representation	Description
0	0000000X	If set, this structure is a directory. If clear, this structure is a file.
1 - 2	00000XX0	The index level of the file – and must contain either a 0, 1, 2, or 3.
3 - 7	XXXXX000	Contains the parent Inode of this file or directory. XXXXX = 31 for the ROOT directory.

File Size Field

This represents the number of bytes occupied by a file. File data can only be stored on the disk in multiples of PGSIZE bytes – however, that doesn't mean all those bytes are real data. A file may have a length of 17 bytes, for instance, so that it would store 16 bytes in the first data block, and one byte in the second data block.
For a directory header, this field is undefined and should be set to zero.

Z502 Disk Structure

Header For Disk or File

Further explanation of the Fields in a Header:

Index Level

Index Level	Description
0	The location "Data Index" points to a single block. This means that, if this is a directory, that directory contains only a single file or a single subdirectory. If this is a file, it means that file contains only a single data block.
1	The location "Data Index" points to a First Level Index. For a directory, the entries in that index in turn point to a directory or file header. For a file, the entries in that index in turn point to data belonging to that file. An unused index element, not pointing to a file or directory must be set to 0.
2	The location "Data Index" points to a Second Level Index. Each of the entries in this second level index point to a First Level Index which act as described above. Again, unused elements must contain a 0.
3	The location "Data Index" points to a Third Level Index. Each of the entries in this third level index point to a Second Level Index which act as described above. Again, unused elements must contain a 0.

Z502 Disk Structure

Header For Disk or File

Index Sector – showing the layout of disk sectors for a single level index

Byte Offset	Bytes in Field	Description
0	2	Holds the sector number for a block in the file, a directory header, or 0
2	2	Holds the sector number for a block in the file, a directory header, or 0
4	2	Holds the sector number for a block in the file, a directory header, or 0
6	2	Holds the sector number for a block in the file, a directory header, or 0
8	2	Holds the sector number for a block in the file, a directory header, or 0
10	2	Holds the sector number for a block in the file, a directory header, or 0
12	2	Holds the sector number for a block in the file, a directory header, or 0
14	2	Holds the sector number for a block in the file, a directory header, or 0

Z502 Disk Structure

Example of File Index Structure

An example for a file having only one data block (alternative 1).

File Header

Field	Value
File / Dir	File
Index Level	0
Index Location	
File Size (Bytes)	<= 16

Z502 Disk Structure

Example of File Index Structure

An example for a file having only one data block (alternative 2).

File Header

Field	Value
File / Dir	File
Index Level	1
Index Location	
File Size (Bytes)	<= 16

Index

Offset	Value
0	
2	0
4	0
6	0
8	0
10	0
12	0
14	0

Z502 Disk Structure

Example of File Index Structure

An example for a file having only one data block (alternative 3).

File Header

Field	Value
File / Dir	File
Index Level	2
Index Location	
File Size (Bytes)	<= 16

Index

Offset	Value
0	
2	0
4	0
6	0
8	0
10	0
12	0
14	0

Index

Offset	Value
0	
2	0
4	0
6	0
8	0
10	0
12	0
14	0

Z502 Disk Structure

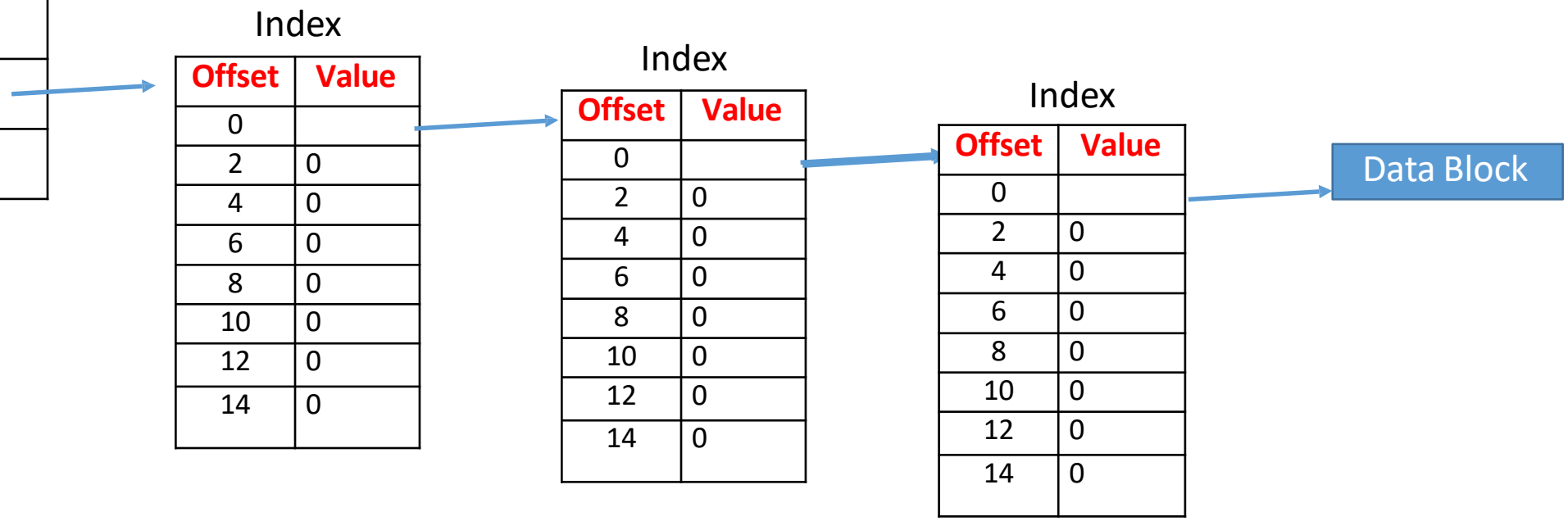
Example of File Index Structure

An example for a file having only one data block (alternative 4).

File Header

Field	Value
File / Dir	File
Index Level	3
Index Location	
File Size (Bytes)	<= 16

The Index Level shows the number of index blocks that must be Accessed before reaching Data Blocks.



Z502 Disk Structure

Example of File Index Structure

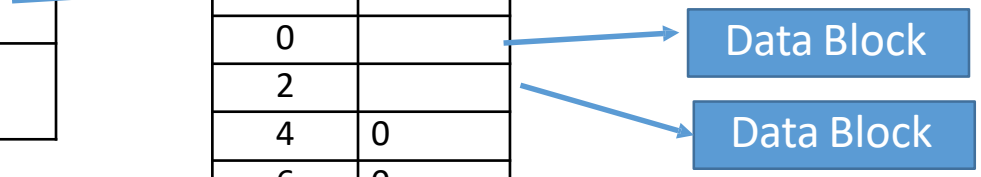
An example for a file having only two data block. (there are many other alternatives)

File Header

Field	Value
File / Dir	File
Index Level	1
Index Location	
File Size (Bytes)	>16, < 32

Index

Offset	Value
0	
2	
4	0
6	0
8	0
10	0
12	0
14	0



Z502 Disk Structure

Example of File Index Structure

An INCORRECT example for a file having only two data block

File Header

Field	Value
File / Dir	File
Index Level	1
Index Location	
File Size (Bytes)	>16, < 32

Index

Offset	Value
0	
2	
4	0
6	0
8	0
10	0
12	0
14	0

Data Block

Index

Offset	Value
0	
2	
4	0
6	0
8	0
10	0
12	0
14	0

Data Block

THIS IS NOT CORRECT. All Data Blocks for a file must traverse the same number of index blocks.

Z502 Disk Structure

Example of File Index Structure

An example for a file having nine data blocks. (there are many other alternatives)

File Header

Field	Value
File / Dir	File
Index Level	2
Index Location	
File Size (Bytes)	> 8 * 16, < 9 * 16

16 is the number of data bytes in a Data Block

Index

Offset	Value
0	
2	
4	0
6	0
8	0
10	0
12	0
14	0

Index

Offset	Value
0	
2	
4	0
6	0
8	0
10	0
12	0
14	0

Data Block
Data Block
Data Block
Data Block
Data Block
Data Block

Index

Offset	Value
0	
2	
4	0
6	0
8	0
10	0
12	0
14	0

Data Block
Data Block
Data Block

While this is technically a legal disk layout, how do you know how to find, say the seventh block, without walking through all the index blocks.

Z502 Disk Structure

Example of Directory Index Structure

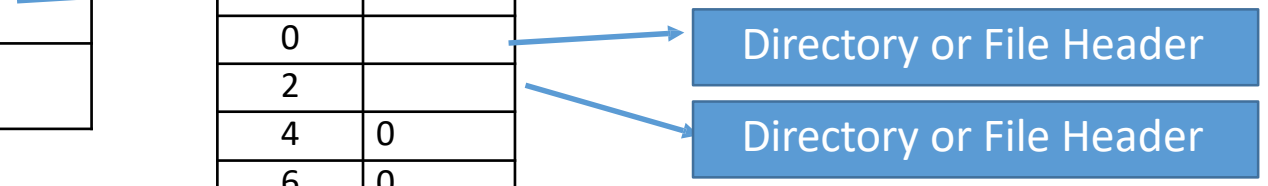
An example for a directory having two File or Directory header blocks. (there are many other alternatives)

File Header

Field	Value
File / Dir	Directory
Index Level	1
Index Location	
File Size (Bytes)	0

Index

Offset	Value
0	
2	
4	0
6	0
8	0
10	0
12	0
14	0



File System Calls FORMAT

Usage of FORMAT

long DiskID;
long ErrorReturned;

FORMAT (DiskID, &ErrorReturned);

Causes the disk with ID DiskID to be formatted by the Operating System. The following operations are accomplished:

1. Sector 0 on the disk is given the structure described previously.
2. A structure for the root directory (with name “root” has been defined and placed on the disk at the Sector pointed at within Sector 0.
3. The bitmap has been defined on the disk, being placed at the location pointed at within Sector 0.
4. The Swap Area is defined and placed on the disk at the location pointed to by Sector 0.
5. All of these auxiliary structures, the bitmap, root directory, and swap area are properly initialized.

The only possible error here is if the DiskID is not in the legal range.

File System Calls CHECK_DISK

Usage of CHECK_DISK

long DiskID;
long ErrorReturned;

CHECK_DISK (DiskID, &ErrorReturned);

This System call causes the Operating System to call a Hardware Function that writes out the contests of the disk into a file named “CheckDiskData.” You are able to print out this file to observe the contents of the disk.

In addition, you can run the program CheckDisk against this file in order to determine if the structure on the disk is correctly managed according to the specification.

The only possible error for the Check_Disk call is if the DiskID is not in the legal range.

File System Calls OPEN_DIR

Usage of OPEN_DIR

```
long  DiskID_OR_Minus1;
char *DirectoryName;
long  ErrorReturned;
```

OPEN_DIR (DiskID_OR_Minus1**,** DirectoryName**, &ErrorReturned);**

This System call causes the Operating System to open a directory on the specified disk. This directory may or may not already exist. If it does NOT already exist, it is created. The directory you have open is defined as your "Current Directory". You can only use this call to open a directory which is a parent or child directory of your Current Directory. To move to a parent directory, you can use a Filename of "..". You can only have one directory open at a time.

Once you have established the Current Directory, you no longer need to explicitly specify the DiskID – this is because further OPEN_DIR calls work from the established Current Directory. You should use a value of "-1" in this case.

This call is a precursor to opening or creating new directories and files. The new entities can only be created in the current directory.

Possible errors for the **OPEN_DIR include:**

- DiskID is not in the legal range.
- The Directory name is illegal.
- There is no room on the disk for the newly created directory.
- If the DirectoryName is specified as ".." and the present directory is "root".

File System Calls

Notes about Current Directory

The Current Directory is a property of a PROCESS.
For instance, two processes could have the same disk open (though only one of those processes would have formatted the disk.) Then each of those processes could have their Current Directory at a different place on the disk and each of them could then operate independently of the other.

In the same way, a process could not, in this architecture, be operating on two disks without closing one of them first.

Notes about Open File

The set of files opened by a program is a property of a PROCESS.

One process could have several files open. Alternatively, two processes, running the same code, could have different files open.

File System Calls OPEN_FILE

Usage of OPEN_FILE

```
char *FileName;
long   Inode;
long  ErrorReturned;
```

OPEN_FILE (FileName, &Inode, &ErrorReturned);

This System call causes the Operating System to open a file in the Current Directory. This file may or may not already exist. If it does NOT already exist, it is created. To Open or Create a file, you must have previously defined a Current Directory by doing an OPEN_DIR System call.

When a file is open, this allows further operations on the file (since there is now an Inode available as a handle.)

Possible errors for the **OPEN_FILE include:**

- The file name is illegal.
- There is no room on the disk for the newly created file.
- There is no previous OPEN_DIR specifying the Current Directory.

File System Calls CREATE_DIR

Usage of CREATE_DIR

char * **DirectoryName**;
long ErrorReturned;

CREATE_DIR (DirectoryName, &ErrorReturned);

This System call causes the Operating System to create a directory of the specified name as a subdirectory of the Current Directory.

The result of this call is to create a Directory. It does NOT open that directory.

Possible errors for the **CREATE_DIR** call include:

- DirectoryName already exists in the current directory.
- There is no space remaining on the disk.
- If the DirectoryName contains illegal characters or more than the allowable number of characters for a name.

File System Calls CREATE_FILE

Usage of CREATE_FILE

char * **FileName**;
long ErrorReturned;

CREATE_FILE (FileName, &ErrorReturned);

This System call causes the Operating System to create a file of the specified name in the Current Directory.

The result of this call is to create a File. It does NOT open that File.

Possible errors for the **CREATE_FILE** call include:

FileName already exists in the current directory.
There is no space remaining on the disk.
If the **FileName** contains illegal characters or more than the allowable number of characters for a name.

File System Calls

READ_FILE

Usage of READ_FILE

```
long   Inode;
long  FileLogicalBlock
char    ReadBuffer[]
long   ErrorReturned;
```

READ_FILE (Inode, FileLogicalBlock, ReadBuffer, &ErrorReturned);

This System call causes the Operating System to read a block of data from the disk, at a location associated with a logical block number of the file having the given Inode.

The Operating System does NOT return control to the user program until it is known that the disk has been read from. At the time control is returned to the user program, the O.S. must have transferred data to the user-defined buffer.

If the FileLogicalBlock is a location on the disk that has not previously been written to the Z502 hardware will cause an error.

Possible errors for the **READ_FILE** call include:

- The Inode that was input does not beto an opened file.
- The FileLogicalBlock is negative or is larger than is supported by the file system.
- The requested block has not been previously written.

File System Calls WRITE_FILE

Usage of WRITE_FILE

long Inode;
long FileLogicalBlock
char WriteBuffer[]
long ErrorReturned;

WRITE_FILE (Inode, FileLogicalBlock, WriteBuffer, &ErrorReturned);

This System call causes the Operating System to write a block of data to the disk, and associate that data with a logical block number in the file having the given Inode.

The Operating System does NOT return control to the user program until it is known that the disk has been written to; in other words, this data is not cached.

Possible errors for the **WRITE_FILE** call include:

- The Inode that was input does not beto an opened file.
- The FileLogicalBlock is negative or is larger than is supported by the file system.
- There is no space remaining on the disk.

File System Calls CLOSE_FILE

Usage of CLOSE_FILE

long Inode;
long ErrorReturned;

CLOSE_FILE(Inode, &ErrorReturned);

This System call causes the Operating System to Close the file. This ensures that all data and metadata has been written to the disk. So, in addition to the DATA that may have previously been written, all indices associated with files, and all bitmap information is now also written to the disk.

Possible errors for the **CLOSE_FILE** call include:

The Inode that was input does not beto an opened file.

File System Calls

DIR_CONTENTS

Usage of DIR_CONTENTS

long ErrorReturned;

DIR_CONTENTS (&ErrorReturned);

This System call causes the Operating System to print information about the directories and files in the Current Directory. The format for this output is as follows (it's essentially all the information in the Directory/File header).

Fields include: Inode, Filename, D/F, Creation Time, Number of Bytes in File.
If you were to execute this System Call on the directory "root" resulting from Test10, you would get the following output.

Contents of Directory root:

Inode,	Filename,	D/F,	Creation Time,	File Size
1	Test10	D	123	--
2	file1	F	145	0
3	file2	F	155	0

Possible errors for the **DIR_CONTENTS** call include:

There is not currently a directory that has been opened and so there is no Current Directory.

File System Calls DELETE_DIR

Usage of DELETE_DIR

char * **DirectoryName**;
long ErrorReturned;

DELETE_DIR (DirectoryName, &ErrorReturned);

This System call causes the Operating System to delete a directory of the specified name in the Current Directory.

The result of this call is to delete the specified Directory.

It is not possible to delete the "root" Directory.

Possible errors for the **DELETE_DIR** call include:

DirectoryName does not exist in the Current Directory.

File System Calls DELETE_FILE

Usage of DELETE_FILE

char * **FileName**;
long ErrorReturned;

DELETE_FILE (FileName, &ErrorReturned);

This System call causes the Operating System to delete a file of the specified name in the Current Directory.

The result of this call is to delete the specified File.

Possible errors for the **DELETE_FILE** call include:

FileName does not exist in the current directory.