Semihosting for AArch32 and AArch64

Release 2.0

ARM

Semihosting for AArch32 and AArch64

Document History

Issue	Date	Confidentiality	Change
2.0	16 December 2016	Non-Confidential	Semihosting extensions incorporated and text updated

Non-Confidential Proprietary Notice

Confidentiality Status

Product Status

The information in this document is Final, that is for a developed product.

Web Address

http://www.arm.com

CHAPTER

ONE

PREFACE

About this document

This document describes the ARM® semihosting mechanism, and its extensions.

Glossary

The ARM Glossary is a list of terms used in ARM documentation, together with definitions for those terms. The ARM Glossary does not contain terms that are industry standard unless the ARM meaning differs from the generally accepted meaning.

See the ARM Glossary for more information.

Feedback

ARM welcomes feedback on this product and its documentation.

Feedback on this product

If you have any comments or suggestions about this product, contact your supplier and give:

- The product name.
- The product revision or version.
- An explanation with as much information as you can provide. Include symptoms and diagnostic procedures if appropriate.

Feedback on content

If you have comments on content then send an e-mail to errata@arm.com. Give:

- The title Semihosting for AArch32 and AArch64.
- If applicable, the page number(s) to which your comments refer.
- A concise explanation of your comments.

ARM also welcomes general suggestions for additions and improvements.

Note: ARM tests the PDF only in Adobe Acrobat and Acrobat Reader, and cannot guarantee the quality of the represented document when used with any other PDF reader.

Other information

- ARM Developer
- ARM Technical Support Knowledge Articles.
- Support and Maintenance.
- ARM Glossary

CHAPTER

TWO

INTRODUCTION

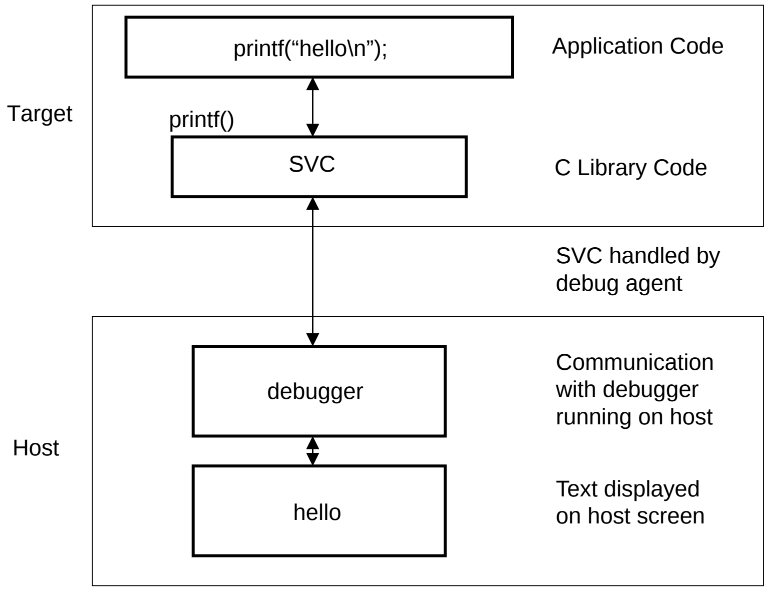
Semihosting is a mechanism that enables code running on an ARM target or emulator to communicate with and use the Input/Output facilities on a host computer. The host must be running the emulator, or a debugger that is attached to the ARM target.

Examples of these facilities include keyboard input, screen output, and disk I/O. For example, you can use this mechanism to enable functions in the C library, such as `printf()` and `scanf()`, to use the screen and keyboard of the host instead of having a screen and keyboard on the target system.

This is useful because development hardware often does not have all the input and output facilities of the final system. Semihosting enables the host computer to provide these facilities.

Semihosting is implemented by a set of defined software instructions that generate exceptions from program control. The application invokes the appropriate semihosting call and the debug agent then handles the exception. The debug agent provides the required communication with the host.

The semihosting interface is common across all debug agents that are provided by ARM. Semihosted operations work when you are debugging applications on your development platform, as shown in the following figure:


printf("hello\n");
Application Code
Target
printf()
SVC
C Library Code
SVC handled by debug agent
debugger
Communication with debugger running on host
Host
hello
Text displayed on host screen

In many cases, semihosting is invoked by code within library functions. The application can also invoke the semihosting operation directly.

Note: The instruction that is used to make semihosting calls can be `SVC`, `HLT`, or `BKPT`, depending on the processor.

See *The semihosting interface* for more detail.

The following terms are used in the semihosting documentation:

Semihosting Implementation An agent (for example, a debugger or emulator) that services semihosting operation requests from a Semihosting Caller executing on an ARM target.

Semihosting Caller A program executing on an ARM target that sends requests to a Semihosting Implementation (for example, a debugger or emulator) to service semihosting operations.

Related information

- *The semihosting interface*
- The ARM C and C++ libraries

CHAPTER

THREE

THE SEMIHOSTING INTERFACE

Semihosting is supported for ARM A and R profiles using the A64, A32, and T32 instruction sets, and for M profile using the T32 instruction set.

Semihosting operations are requested using a trap instruction, which is a software instruction that generates exceptions from program control. Semihosting callers issue trap instructions, and the semihosting implementation then handles the resulting exception to perform the required semihosting operation. The trap instruction can be `SVC`, `HLT`, or `BKPT`, as indicated in Table 3.1:

Table 3.1: Semihosting Trap Instructions and Encodings

Profile	Instruction Set	Instruction	Opcode
A+R Profile	A64	`HLT #0xF000`	`0xD45E0000`
	A32	`SVC #0x123456`	`0xEF123456`
		`HLT #0xF000`	`0xE10F0070`
	T32	`SVC #0xAB`	`0xDFAB`
		`HLT #0x3C`	`0xBABC`
M–Profile	T32	`BKPT #0xAB`	`0xBEAB`

For A32 and T32 on A+R Profile, semihosting can use either an `SVC` or an `HLT` trap instruction. Semihosting implementations must support both trap instructions across all versions of the ARM architecture.

Note: This requirement includes supporting the `HLT` encodings on ARMv7 and earlier processors, even though `HLT` is only defined as an instruction in ARMv8. This may require the semihosting implementation to trap the `UNDEF` exception.

The `HLT` encodings are new in version 2.0 of the semihosting specification. Where possible, have semihosting callers continue to use the previously existing trap instructions to ensure compatibility with legacy semihosting implementations. These trap instructions are `HLT` for A64, `SVC` on A+R profile A32 or T32, and `BKPT` on M profile. However, it is necessary to change from SVC to HLT instructions to support AArch32 semihosting properly in a mixed AArch32/AArch64 system.

Note: ARM encourages semihosting callers to implement support for trapping using `HLT` on A32 and T32 as a configurable option. ARM strongly discourages semihosting callers from mixing the `HLT` and `SVC` mechanisms within the same executable.

The OPERATION NUMBER REGISTER must contain the number of the semihosting operation to be performed. This number is given in parentheses after the operation name in the following sections. For example, `SYS_OPEN (0x01)`.

Parameters are passed to the operation using the PARAMETER REGISTER. For most operations, the PARAMETER

REGISTER must contain a pointer to a data block that holds the parameters. In some cases a single parameter is passed directly in the PARAMETER REGISTER, or no parameters are passed at all.

Results are returned in the RETURN REGISTER, either as an explicit return value or as a pointer to a data block. If no result is returned, assume that the RETURN REGISTER is corrupted. Some operations also return information in the PARAMETER REGISTER.

Multi-byte values in memory must be formatted as pure little-endian or pure big-endian to match the endianness mapping configuration of the processor.

Table 3.2 shows the specific registers that are used, and the size of the fields in the data block, which depend on whether the caller is 32-bit or 64-bit.

Table 3.2: Registers and field size

	32-bit	64-bit
OPERATION NUMBER REGISTER	`R0`	`W0`
PARAMETER REGISTER	`R1`	`X1`
RETURN REGISTER	`R0`	`X0`
Data block field size	32 bits	64 bits

Note: The operation number is passed in `W0` for the 64-bit ABI, which is the bottom 32 bits of the 64-bit register `X0`. Semihosting implementations must not assume that the top 32 bits of `X0` are 0.

A few semihosting operations have other differences between the 32-bit and 64-bit versions. These differences are described in the documentation of those operations.

The available semihosting operation numbers are allocated as follows:

`0x00-0x31` Used by ARM.

`0x32-0xFF` Reserved for future use by ARM.

`0x100-0x1FF` Reserved for user applications. These semihosting operation numbers are not used by ARM. However if you are writing your own `SVC` operations, you are advised to use a different `SVC` number, rather than using the semihosted `SVC` number and these operation type numbers.

`0x200-0xFFFFFFFF` Undefined and currently unused. ARM recommends that you do not use these semihosting operation numbers.

Note: Previous versions of the semihosting specification included the operation numbers `0x17` (`angel_SWIreason_EnterSVC`) and `0x19` (`angelSWI_Reason_SyncCacheRange`). These semihosting operation numbers are now reserved, and are not to be used or implemented.

Related information

- A32 and T32 instruction sets
- A64 instruction set

CHAPTER

FOUR

SEMIHOSTING EXTENSIONS

The semihosting API provides an extension mechanism. Semihosting implementations can provide optional functionality and advertise to the semihosting caller that they do so. Semihosting callers must check that the implementation supports the optional functionality before attempting to use it.

Each extension that the specification defines has an associated feature bit. If the implementation provides the extension, then it reports the feature bit as 1. Each extension definition identifies the feature bit by which feature byte it is in, and which bit within that byte. For example, feature byte 0, bit 3. Feature bytes and bits within them are both numbered starting from 0, with feature bit 0 being the least significant bit. Extensions are independent. Unless otherwise stated, support for one feature does not imply support for any other feature. The caller must check the feature bit for every feature it wants to use.

Semihosting feature bit reporting sequence format

Feature bits are reported using a sequence of bytes, which are accessed by using the `SYS_OPEN` call with the special path name `:semihosting-features`. The byte sequence has the following format:

```
byte 0: SHFB_MAGIC_0 0x53
byte 1: SHFB_MAGIC_1 0x48
byte 2: SHFB_MAGIC_2 0x46
byte 3: SHFB_MAGIC_3 0x42
byte 4: feature byte 0
byte 5: feature byte 1
byte 6: feature byte 2
...
```

There is no limit to the number of bytes that can be returned. As future extensions are defined, new bytes will be added. If the read reaches the end of the file before the byte that contains a particular feature bit, then that feature bit must be taken to be 0.

Since the file is a sequence of bytes, the order remains the same whether the processor is big-endian or little-endian. The contents must be treated as a byte array, not an array of words or any other larger type.

Requirements for semihosting implementations

An implementation that implements any semihosting extensions must do the following:

- Handle the special path name `:semihosting-features` in `SYS_OPEN` when opened with mode 0 (`r`) or 1 (`rb`). Implement it to return a filehandle that behaves as if it were accessing a file containing a sequence of bytes in the format that is described in *Semihosting feature bit reporting sequence format*. Attempts to open this special path name with any other opening mode must fail. The implementation must support opening of this special path name multiple times, both consecutively and simultaneously.

- Support the following operations on the filehandle returned from `SYS_OPEN` of `:semihosting-features` :

  `SYS_FLEN` Returns the length of the byte sequence, including the magic number bytes.

  `SYS_ISTTY` Always returns 0.

  `SYS_SEEK` Permits random seeks to anywhere within the byte sequence.

  `SYS_READ` Reads bytes from the sequence starting from the current seek position.

  `SYS_CLOSE` Closes the filehandle.

The special path name does not correspond to a real file in a filesystem, and so no special case handling of it is expected for `SYS_REMOVE` or `SYS_RENAME`.

Requirements for semihosting callers

To read the feature bits, the semihosting caller must:

1. Call `SYS_OPEN` with the special filename `:semihosting-features` and the file opening mode 0 (`r`). If the `SYS_OPEN` fails, then no extensions are supported, so all feature bits should be assumed to be 0.
2. Use the `SYS_READ` call to read bytes from the filehandle returned by step 1. If the read returns fewer than 5 bytes, or the first 4 bytes are not the correct magic numbers, then no extensions are supported.

   The caller must check all the magic bytes, but can optionally use `SYS_SEEK` to seek forwards in the byte sequence to the subsequent byte, or bytes, of interest. Seeking off the end of the file is undefined, so a caller wanting to use `SYS_SEEK` must first use `SYS_FLEN` to determine the size of the byte sequence.
3. If the `SYS_OPEN` from step 1 did not fail, the caller must now use `SYS_CLOSE` to close the filehandle.

It is important to follow this sequence to ensure correct behavior on legacy semihosting implementations, which do not recognize the special filename `:semihosting-features`.

Example pseudocode function for querying feature bits

The following C-like pseudocode describes one possible simple implementation of a check for a specific feature bit:

```
#define MAGICLEN 4
bool sh_feature_supported(int bytenum, int bitnum)
{
    unsigned char magic[MAGICLEN];
    unsigned char c;
    int fh;
    int len;

    fh = sys_open(":semihosting_features", 0);
    if (fh == -1) {
        return false;
    }
    len = sys_flen(fh);
    if (len <= bytenum) {
        sys_close(fh);
        return false;
    }
    if (sys_read(fh, &magic, MAGICLEN) != 0) {
        sys_close(fh);
        return false;
    }
    if (magic[0] != SHFB_MAGIC_0 ||
        magic[1] != SHFB_MAGIC_1 ||
        magic[2] != SHFB_MAGIC_2 ||
        magic[3] != SHFB_MAGIC_3) {
        sys_close(fh);
        return false;
    }
    if (sys_seek(fh, bytenum) != 0) {
        sys_close(fh);
        return false;
    }
    if (sys_read(fh, &c, 1) != 0) {
        sys_close(fh);
        return false;
    }
    sys_close(fh);
    return (c & (1 << bitnum)) != 0;
}
```

More sophisticated implementations can choose to cache the contents of the start of the file with the magic bytes and the first few bytes of feature bit data. Conversely if, for instance, the caller knows in advance that it is only interested in feature bits inside the first feature byte, it can simplify this method to perform a single 5 byte read and need not call `SYS_FLEN` or `SYS_SEEK`.

Semihosting extensions available

Table 4.1 lists the semihosting extensions currently available, along with their associated feature byte and bit.

Table 4.1: Semihosting Extensions

Name	Feature byte	Feature bit
SH_EXT_EXIT_EXTENDED	0	0
SH_EXT_STDOUT_STDERR	0	1

CHAPTER FIVE

SEMIHOSTING OPERATIONS

The following semihosting operations are available:

- *SYS_CLOCK (0x10)*
- *SYS_CLOSE (0x02)*
- *SYS_ELAPSED (0x30)*
- *SYS_ERRNO (0x13)*
- *SYS_EXIT (0x18)*
- *SYS_EXIT_EXTENDED (0x20)*
- *SYS_FLEN (0x0C)*
- *SYS_GET_CMDLINE (0x15)*
- *SYS_HEAPINFO (0x16)*
- *SYS_ISERROR (0x08)*
- *SYS_ISTTY (0x09)*
- *SYS_OPEN (0x01)*
- *SYS_READ (0x06)*
- *SYS_READC (0x07)*
- *SYS_REMOVE (0x0E)*
- *SYS_RENAME (0x0F)*
- *SYS_SEEK (0x0A)*
- *SYS_SYSTEM (0x12)*
- *SYS_TICKFREQ (0x31)*
- *SYS_TIME (0x11)*
- *SYS_TMPNAM (0x0D)*
- *SYS_WRITE (0x05)*
- *SYS_WRITEC (0x03)*
- *SYS_WRITE0 (0x04)*

SYS_CLOCK (0x10)

Returns the number of centiseconds (hundredths of a second) since the execution started.

Values returned can be of limited use for some benchmarking purposes because of communication overhead or other agent-specific factors. For example, with a debug hardware unit the request is passed back to the host for execution. This can lead to unpredictable delays in transmission and process scheduling.

Use this function to calculate time intervals, by calculating differences between intervals with and without the code sequence to be timed.

Entry

The PARAMETER REGISTER must contain 0. There are no other parameters.

Return

On exit, the RETURN REGISTER contains:

- The number of centiseconds since some arbitrary start point, if the call is successful.
- –1 if the call is not successful. For example, because of a communications error.

Related information

- *SYS_ELAPSED (0x30)*
- *SYS_TICKFREQ (0x31)*

SYS_CLOSE (0x02)

Closes a file on the host system. The handle must reference a file that was opened with `SYS_OPEN`.

Entry

On entry, the PARAMETER REGISTER contains a pointer to a one-field argument block:

field 1 Contains a handle for an open file.

Return

On exit, the RETURN REGISTER contains:

- 0 if the call is successful
- –1 if the call is not successful.

Related information

- *SYS_OPEN (0x01)*

`SYS_ELAPSED (0x30)`

Returns the number of elapsed target ticks since execution started.

Use `SYS_TICKFREQ` to determine the tick frequency.

Entry (32-bit)

On entry, the PARAMETER REGISTER points to a two-field data block to be used for returning the number of elapsed ticks:

field 1 The least significant field and is at the low address.

field 2 The most significant field and is at the high address.

Entry (64-bit)

On entry the PARAMETER REGISTER points to a one-field data block to be used for returning the number of elapsed ticks:

field 1 The number of elapsed ticks as a 64-bit value.

Return

On exit:

- On success, the RETURN REGISTER contains 0, the PARAMETER REGISTER is unchanged, and the data block pointed to by the PARAMETER REGISTER is filled in with the number of elapsed ticks.
- On failure, the RETURN REGISTER contains -1, and the PARAMETER REGISTER contains -1.

Note: Some semihosting implementations might not support this semihosting operation, and they always return -1 in the RETURN REGISTER.

Related information

- *SYS_TICKFREQ (0x31)*

SYS_ERRNO (0x13)

Returns the value of the C library `errno` variable that is associated with the semihosting implementation.

The `errno` variable can be set by a number of C library semihosted functions, including:

- `SYS_REMOVE`
- `SYS_OPEN`
- `SYS_CLOSE`
- `SYS_READ`
- `SYS_WRITE`
- `SYS_SEEK`.

Whether `errno` is set or not, and to what value, is entirely host-specific, except where the ISO C standard defines the behavior.

Entry

There are no parameters. The PARAMETER REGISTER must be 0.

Return

On exit, the RETURN REGISTER contains the value of the C library `errno` variable.

Related information

- *SYS_CLOSE (0x02)*
- *SYS_OPEN (0x01)*
- *SYS_READ (0x06)*
- *SYS_REMOVE (0x0E)*
- *SYS_SEEK (0x0A)*
- *SYS_WRITE (0x05)*

`SYS_EXIT (0x18)`

Note: `SYS_EXIT` was called `angel_SWIreason_ReportException` in previous versions of the documentation.

An application calls this operation to report an exception to the debugger directly. The most common use is to report that execution has completed, using `ADP_Stopped_ApplicationExit`.

Note: This semihosting operation provides no means for 32-bit callers to indicate an application exit with a specified exit code. Semihosting callers may prefer to check for the presence of the `SH_EXT_EXTENDED_REPORT_EXCEPTION` extension and use the `SYS_REPORT_EXCEPTION_EXTENDED` operation instead, if it is available.

Entry (32-bit)

On entry, the PARAMETER register is set to a reason code describing the cause of the trap. Not all semihosting client implementations will necessarily trap every corresponding event. These reason codes are defined in the following tables.

The following table shows reason codes relating to hardware exceptions. Exception handlers can use these operations to report an exception that has not been handled:

Name	Hexadecimal value
`ADP_Stopped_BranchThroughZero`	`0x20000`
`ADP_Stopped_UndefinedInstr`	`0x20001`
`ADP_Stopped_SoftwareInterrupt`	`0x20002`
`ADP_Stopped_PrefetchAbort`	`0x20003`
`ADP_Stopped_DataAbort`	`0x20004`
`ADP_Stopped_AddressException`	`0x20005`
`ADP_Stopped_IRQ`	`0x20006`
`ADP_Stopped_FIQ`	`0x20007`

The following table shows reason codes relating to software events:

Name	Hexadecimal value
`ADP_Stopped_BreakPoint`	`0x20020`
`ADP_Stopped_WatchPoint`	`0x20021`
`ADP_Stopped_StepComplete`	`0x20022`
`ADP_Stopped_RunTimeErrorUnknown`	`0x20023`
`ADP_Stopped_InternalError`	`0x20024`
`ADP_Stopped_UserInterruption`	`0x20025`
`ADP_Stopped_ApplicationExit`	`0x20026`
`ADP_Stopped_StackOverflow`	`0x20027`
`ADP_Stopped_DivisionByZero`	`0x20028`
`ADP_Stopped_OSSpecific`	`0x20029`

Entry (64-bit)

On entry, the PARAMETER REGISTER contains a pointer to a two-field argument block:

field 1 The exception type, which is one of the set of reason codes in the above tables.

field 2 A subcode, whose meaning depends on the reason code in field 1.

In particular, if field 1 is `ADP_Stopped_ApplicationExit` then field 2 is an exit status code, as passed to the C standard library `exit()` function. A simulator receiving this request must notify a connected debugger, if present, and then exit with the specified status.

Return

No return is expected from these calls. However, it is possible for the debugger to request that the application continues by performing an `RDI_Execute` request or equivalent. In this case, execution continues with the registers as they were on entry to the operation, or as subsequently modified by the debugger.

Related information

- *SYS_EXIT_EXTENDED (0x20)*

SYS_EXIT_EXTENDED (0x20)

This operation is only supported if the semihosting extension SH_EXT_EXIT_EXTENDED is implemented. SH_EXT_EXIT_EXTENDED is reported using feature byte 0, bit 0. If this extension is supported, then the implementation provides a means to report a normal exit with a nonzero exit status in both 32-bit and 64-bit semihosting APIs.

The implementation must provide the semihosting call SYS_EXIT_EXTENDED for both A64 and A32/T32 semihosting APIs.

SYS_EXIT_EXTENDED is used by an application to report an exception or exit to the debugger directly. The most common use is to report that execution has completed, using ADP_Stopped_ApplicationExit.

Entry

On entry, the PARAMETER REGISTER contains a pointer to a two-field argument block:

field 1 The exception type, which should be one of the set of reason codes that are documented for the SYS_EXIT (0x18) call. For example, ADP_Stopped_ApplicationExit or ADP_Stopped_InternalError.

field 2 A subcode, whose meaning depends on the reason code in field 1. In particular, if field 1 is ADP_Stopped_ApplicationExit then field 2 is an exit status code, as passed to the C standard library exit() function. A simulator receiving this request must notify a connected debugger, if present, and then exit with the specified status.

Return

No return is expected from these calls.

For the A64 API, this call is identical to the behavior of the mandatory SYS_EXIT (0x18) call. If this extension is supported, then both calls must be implemented.

Related information

- *SYS_EXIT (0x18)*

SYS_FLEN (0x0C)

Returns the length of a specified file.

Entry

On entry, the PARAMETER REGISTER contains a pointer to a one-field argument block:

field 1 A handle for a previously opened, seekable file object.

Return

On exit, the RETURN REGISTER contains:

- The current length of the file object, if the call is successful.
- –1 if an error occurs.

`SYS_GET_CMDLINE (0x15)`

Returns the command line that is used for the call to the executable, that is, `argc` and `argv`.

Entry

On entry, the PARAMETER REGISTER points to a two-field data block to be used for returning the command string and its length:

field 1 A pointer to a buffer of at least the size that is specified in field 2.

field 2 The length of the buffer in bytes.

Return

On exit:

If the call is successful, then the RETURN REGISTER contains 0, the PARAMETER REGISTER is unchanged, and the data block is updated as follows:

field 1 A pointer to a null-terminated string of the command line.

field 2 The length of the string in bytes.

If the call is not successful, then the RETURN REGISTER contains -1.

Note: The semihosting implementation might impose limits on the maximum length of the string that can be transferred. However, the implementation must be able to support a command-line length of at least 80 bytes.

SYS_HEAPINFO (0x16)

Returns the system stack and heap parameters.

Entry

On entry, the PARAMETER REGISTER contains the address of a pointer to a four-field data block. The contents of the data block are filled by the function. The following C-like pseudocode describes the layout of the block:

```
struct block {
    void* heap_base;
    void* heap_limit;
    void* stack_base;
    void* stack_limit;
};
```

Return

On exit, the PARAMETER REGISTER is unchanged and the data block has been updated.

SYS_ISERROR (0x08)

Determines whether the return code from another semihosting call is an error status or not.

This call is passed a parameter block containing the error code to examine.

Entry

On entry, the PARAMETER REGISTER contains a pointer to a one-field data block:

field 1 The required status word to check.

Return

On exit, the RETURN REGISTER contains:

- 0 if the status field is not an error indication
- A nonzero value if the status field is an error indication.

SYS_ISTTY (0x09)

Checks whether a file is connected to an interactive device.

Entry

On entry, the PARAMETER REGISTER contains a pointer to a one-field argument block:

field 1 A handle for a previously opened file object.

Return

On exit, the RETURN REGISTER contains:

- 1 if the handle identifies an interactive device.
- 0 if the handle identifies a file.
- A value other than 1 or 0 if an error occurs.

SYS_OPEN (0x01)

Opens a file on the host system.

The file path is specified either as relative to the current directory of the host process, or absolute, using the path conventions of the host operating system.

Semihosting implementations must support opening the special path name `:semihosting-features` as part of the semihosting extensions reporting mechanism. See *Semihosting extensions* for details.

ARM targets interpret the special path name `:tt` as meaning the console input stream, for an open-read or the console output stream, for an open-write. Opening these streams is performed as part of the standard startup code for those applications that reference the C `stdio` streams. The semihosting extension `SH_EXT_STDOUT_STDERR` allows the semihosting caller to open separate output streams corresponding to `stdout` and `stderr`. This extension is reported using feature byte 0, bit 1. Use `SYS_OPEN` with the special path name `:semihosting-features` to access the feature bits. See *Semihosting extensions* for details.

If this extension is supported, the implementation must support the following additional semantics to `SYS_OPEN`:

- If the special path name `:tt` is opened with an `fopen` mode requesting write access (`w`, `wb`, `w+`, or `w+b`), then this is a request to open `stdout`.
- If the special path name `:tt` is opened with a mode requesting append access (`a`, `ab`, `a+`, or `a+b`), then this is a request to open `stderr`.

Entry

On entry, the PARAMETER REGISTER contains a pointer to a three-field argument block:

field 1 A pointer to a null-terminated string containing a file or device name.

field 2 An integer that specifies the file opening mode. Table 5.1 gives the valid values for the integer, and their corresponding ISO C `fopen()` mode.

field 3 An integer that gives the length of the string pointed to by field 1.

The length does not include the terminating null character that must be present.

Table 5.1: Value of mode

Mode	0	1	2	3	4	5	6	7	8	9	10	11
ISO C fopen mode [1]	r	rb	r+	r+b	w	wb	w+	w+b	a	ab	a+	a+b

Return

On exit, the RETURN REGISTER contains:

- A nonzero handle if the call is successful.
- –1 if the call is not successful.

[1] The non-ANSI option `t` is not supported.

`SYS_READ (0x06)`

Reads the contents of a file into a buffer.

The file position is specified either:

- Explicitly by a `SYS_SEEK`.
- Implicitly one byte beyond the previous `SYS_READ` or `SYS_WRITE` request.

The file position is at the start of the file when it is opened, and is lost when the file is closed. Perform the file operation as a single action whenever possible. For example, do not split a read of 16KB into four 4KB chunks unless there is no alternative.

Entry

On entry, the PARAMETER REGISTER contains a pointer to a three-field data block:

field 1 Contains a handle for a file previously opened with `SYS_OPEN`.

field 2 Points to a buffer.

field 3 Contains the number of bytes to read to the buffer from the file.

Return

On exit, the RETURN REGISTER contains the number of bytes not filled in the buffer (`buffer_length -bytes_read`) as follows:

- If the RETURN REGISTER is 0, the entire buffer was successfully filled.
- If the RETURN REGISTER is the same as field 3, no bytes were read (EOF can be assumed).
- If the RETURN REGISTER contains a value smaller than field 3, the read succeeded but the buffer was only partly filled. For interactive devices, this is the most common return value.

Related information

- *SYS_READC (0x07)*

SYS_READC (0x07)

Reads a byte from the console.

Entry

The PARAMETER REGISTER must contain 0. There are no other parameters or values possible.

Return

On exit, the RETURN REGISTER contains the byte read from the console.

Related information

- *SYS_READ (0x06)*

`SYS_REMOVE (0x0E)`

Deletes a specified file on the host filing system.

Entry

On entry, the PARAMETER REGISTER contains a pointer to a two-field argument block:

field 1 Points to a null-terminated string that gives the path name of the file to be deleted.

field 2 The length of the string.

Return

On exit, the RETURN REGISTER contains:

- 0 if the delete is successful
- A nonzero, host-specific error code if the delete fails.

`SYS_RENAME (0x0F)`

Renames a specified file.

Entry

On entry, the PARAMETER REGISTER contains a pointer to a four-field data block:

field 1 A pointer to the name of the old file.

field 2 The length of the old filename.

field 3 A pointer to the new filename.

field 4 The length of the new filename.

Both strings are null-terminated.

Return

On exit, the RETURN REGISTER contains:

- 0 if the rename is successful.
- A nonzero, host-specific error code if the rename fails.

`SYS_SEEK (0x0A)`

Seeks to a specified position in a file using an offset specified from the start of the file.

The file is assumed to be a byte array and the offset is given in bytes.

Entry

On entry, the PARAMETER REGISTER contains a pointer to a two-field data block:

field 1 A handle for a seekable file object.

field 2 The absolute byte position to seek to.

Return

On exit, the RETURN REGISTER contains:

- 0 if the request is successful.
- A negative value if the request is not successful. Use `SYS_ERRNO` to read the value of the host `errno` variable describing the error.

Note: The effect of seeking outside the current extent of the file object is undefined.

Related information

- *SYS_ERRNO (0x13)*

SYS_SYSTEM (0x12)

Passes a command to the host command-line interpreter.

This enables you to execute a system command such as `dir`, `ls`, or `pwd`. The terminal I/O is on the host, and is not visible to the target.

Caution: The command that is passed to the host is executed on the host. Ensure that any command passed has no unintended consequences.

Entry

On entry, the PARAMETER REGISTER contains a pointer to a two-field argument block:

field 1 Points to a string to be passed to the host command-line interpreter.

field 2 The length of the string.

Return

On exit, the RETURN REGISTER contains the return status.

`SYS_TICKFREQ (0x31)`

Returns the tick frequency.

Entry

The PARAMETER REGISTER must contain 0 on entry to this routine.

Return

On exit, the RETURN REGISTER contains either:

- The number of ticks per second.
- –1 if the target does not know the value of one tick.

Note: Some semihosting implementations might not support this semihosting operation, and they always return -1 in the RETURN REGISTER.

`SYS_TIME (0x11)`

Returns the number of seconds since 00:00 January 1, 1970.

This value is real-world time, regardless of any debug agent configuration.

Entry

There are no parameters.

Return

On exit, the RETURN REGISTER contains the number of seconds.

SYS_TMPNAM (0x0D)

Returns a temporary name for a file identified by a system file identifier.

Entry

On entry, the PARAMETER REGISTER contains a pointer to a three-word argument block:

field 1 A pointer to a buffer.

field 2 A target identifier for this filename. Its value must be an integer in the range 0-255.

field 3 Contains the length of the buffer. The length must be at least the value of `L_tmpnam` on the host system.

Return

On exit, the RETURN REGISTER contains:

- 0 if the call is successful.
- –1 if an error occurs.

The buffer pointed to by the PARAMETER REGISTER contains the filename, prefixed with a suitable directory name.

If you use the same target identifier again, the same filename is returned.

Note: The returned string must be null-terminated.

`SYS_WRITE (0x05)`

Writes the contents of a buffer to a specified file at the current file position.

The file position is specified either:

- Explicitly, by a `SYS_SEEK`.
- Implicitly as one byte beyond the previous `SYS_READ` or `SYS_WRITE` request.

The file position is at the start of the file when the file is opened, and is lost when the file is closed.

Perform the file operation as a single action whenever possible. For example, do not split a write of 16KB into four 4KB chunks unless there is no alternative.

Entry

On entry, the PARAMETER REGISTER contains a pointer to a three-field data block:

field 1 Contains a handle for a file previously opened with `SYS_OPEN`.

field 2 Points to the memory containing the data to be written.

field 3 Contains the number of bytes to be written from the buffer to the file.

Return

On exit, the RETURN REGISTER contains:

- 0 if the call is successful.
- The number of bytes that are not written, if there is an error.

Related information

- *SYS_WRITE0 (0x04)*
- *SYS_WRITEC (0x03)*

`SYS_WRITEC (0x03)`

Writes a character byte, pointed to by the PARAMETER REGISTER, to the debug channel.

When executed under an ARM debugger, the character appears on the host debugger console.

Entry

On entry, the PARAMETER REGISTER contains a pointer to the character.

Return

None. The RETURN REGISTER is corrupted.

Related information

- *SYS_WRITE (0x05)*
- *SYS_WRITE0 (0x04)*

`SYS_WRITE0 (0x04)`

Writes a null-terminated string to the debug channel.

When executed under an ARM debugger, the characters appear on the host debugger console.

Entry

On entry, the PARAMETER REGISTER contains a pointer to the first byte of the string.

Return

None. The RETURN REGISTER is corrupted.

Related information

- *SYS_WRITE (0x05)*
- *SYS_WRITEC (0x03)*