

IMPORTANT NOTICE

The Processors Wiki was decommissioned on January 15, 2021.

Wiki pages that were migrated are redirecting to the new location. Pages not migrated are only accessible to users on the TI network. Users off the network can not access the Wiki and will receive a 404 error. **DO NOT** share Wiki URLs with customers. They can not access the Wiki site.

CIO System Call Protocol

From Texas Instruments Wiki

Jump to: [navigation](#), [search](#)

Contents

- [1 Introduction](#)
- [2 Protocol](#)
 - [2.1 DTOPEN](#)
 - [2.1.1 File descriptors](#)
 - [2.2 DTCLOSE](#)
 - [2.3 DTREAD](#)
 - [2.4 DTWRITE](#)
 - [2.5 DTLSEEK](#)
 - [2.6 DTUNLINK](#)
 - [2.7 DTRENAME](#)
 - [2.8 DTGETENV](#)
 - [2.9 DTTGETIME](#)
 - [2.10 DTTGETIME64](#)
 - [2.11 DTGETCLOCK](#)
- [3 Examples](#)
 - [3.1 C6000 little-endian write](#)
 - [3.1.1 target request](#)
 - [3.1.2 external agent response](#)
 - [3.2 C6000 big-endian write](#)
 - [3.2.1 target request](#)
 - [3.3 MSP little-endian rename](#)

- [3.3.1 target request](#)
- [3.4 C2000 little-endian rename](#)
 - [3.4.1 target request](#)
- [4 TI's implementation](#)

Introduction[[edit](#)]

This document describes the TI CIO system call protocol supported by [Code Composer Studio](#) (CCS) and other debugging tools. The CIO protocol (for C I/O) provides I/O system calls to allow the target (DSP or microcontroller) to access host files. The low-level unix-like functions open, close, read, write, lseek, unlink, rename, getenv, gettimeofday, and clock (collectively called the "low-level" functions in the [TI compiler user's guide](#)) will use CIO system calls. Higher-level standard C functions use the low-level functions and are not aware of the CIO protocol.

CIO is a stop-mode interface; it can only function when the program is under the control of an external agent that supports CIO, typically a debugger (e.g. CCS) on a workstation. If a program uses CIO when such an external agent is not attached, the behavior is undefined. The likely outcome will be that the system calls will do nothing and will have garbage return values. If C I/O is desired for a standalone target, the developer is expected to provide replacements for the I/O functions, and need not follow the CIO protocol.

Protocol[[edit](#)]

There are two specially-named "magic" object file symbols, C\$\$EXIT and C\$\$IO\$\$\$. The external agent is expected to [set a breakpoint](#) at both of these labels. Both labels are optional, but if C\$\$EXIT is not present, the program cannot exit normally, and if C\$\$IO\$\$\$ is not present, no C I/O can be performed.

If target execution reaches C\$\$EXIT, target execution is halted, and does not resume. The CIO interface does not at this time provide a means of passing the target process's exit status to the external agent. C\$\$EXIT is expected to be reached by any means of program termination, such as calling exit or abort or getting an uncaught C++ exception.

If target execution reaches C\$\$IO\$\$\$, the target has requested C I/O, and the external agent must perform it. The external agent halts the target, performs the request, writes the result to target memory, and resumes execution at address C\$\$IO\$\$\$. It is advised to make the instruction at address C\$\$IO\$\$\$ a NOP.

The external agent must decode the contents of the special CIO buffer located at the linker symbol _CIOBUF_. (That's one leading and one trailing underscore.) (It is an error if one of C\$\$IO\$\$\$ and _CIOBUF_ is defined but not the other.) This buffer contains the encoded C I/O request from the target. This buffer is normally placed in the section .cio, but this is not a requirement. The CIO buffer is of size CIOBUFSIZ target chars and

aligned to the alignment of a target int. CIOBUFSIZ must be 288 (256+32) or greater. By default, CIOBUFSIZ is 288. The external agent is expected to handle CIO buffer requests of arbitrary size. If the external agent maintains its own copy of the CIO buffer, it must deduce the size needed from the total size of the current and previous CIO requests, which may require reallocation of the external agent's buffer.

All types used here are in terms of target types, and the term "byte" means what it means in the C standard: one byte is exactly the same size as one target "char." For some TI targets, a byte is 8 bits; however, for C2000 and C5000, one byte is 16 bits. This presents special challenges when using fread; see application note SPRA757 "Reading and Writing Binary Files on Targets With More Than 8-Bit Chars" for implications to the user.

The CIO buffer has the following fields in this order:

1. The length in target bytes of the variable-length data (only), encoded as a target int in target native endianness. The total length of the CIO buffer is implicit. The external agent must know the size of target int and the target endianness beforehand.
2. One byte representing the specific system call command. (This field is omitted from the CIO buffer when the external agent responds, moving the next two fields back one byte.)
3. A parameter buffer of exactly 8 chars. This encodes the fixed-sized arguments for the system call.
4. The variable-length data. This encodes any variable-length arguments for the system call, such as filenames or strings.

The structure of the CIO buffer may be represented as the following C struct (which uses the "struct hack"). When the target makes a system call, the CIO buffer has the following structure:

```
struct request
{
    int length;
    unsigned char command;
    unsigned char parmbuf[8];
    unsigned char data[1];
};
```

When the external agent makes a response, the CIO buffer has the following structure:

```
struct response
{
    int length;
    unsigned char parmbuf[8];
    unsigned char data[1];
};
```

Not every system call uses all of the fields. If unused, the length field must be 0. The other fields need not be initialized if unused.

The encodings for the system calls are:

<code>_DTOPEN</code>	<code>0xF0</code>
<code>_DTCLOSE</code>	<code>0xF1</code>
<code>_DTREAD</code>	<code>0xF2</code>
<code>_DTWRITE</code>	<code>0xF3</code>
<code>_DTLSEEK</code>	<code>0xF4</code>
<code>_DTUNLINK</code>	<code>0xF5</code>
<code>_DTGETENV</code>	<code>0xF6</code>
<code>_DTRENAME</code>	<code>0xF7</code>
<code>_DTGETTIME</code>	<code>0xF8</code>
<code>_DTGETCLK</code>	<code>0xF9</code>
<code>_DTGETTIME64</code>	<code>0xFA</code> (future)
<code>_DTSYNC</code>	<code>0xFF</code> (unused)

`_DTOPEN`[\[edit\]](#)

`open` takes arguments `pathname`, `flags`, and `mode`. (However, the external agent will ignore the `mode` argument.) `open` will format the CIO buffer as a `_DTOPEN` system call that will look like this:

- `length`: `(int)strlen(pathname)+1`
- `command`: `_DTOPEN` (1 byte)
- `parmbuf 0-1`: `mode` (2 bytes, little-endian)
- `parmbuf 2-3`: `flags` (2 bytes, little-endian)
- `parbbuf 4-7`: padding (4 bytes)
- `data`: `pathname` (one ASCII character per byte)

The flags are:

<code>O_RDONLY</code>	<code>0x0000</code>
<code>O_WRONLY</code>	<code>0x0001</code>
<code>O_RDWR</code>	<code>0x0002</code>
<code>O_APPEND</code>	<code>0x0008</code>
<code>O_CREAT</code>	<code>0x0200</code>
<code>O_TRUNC</code>	<code>0x0400</code>
<code>O_BINARY</code>	<code>0x8000</code>

After formatting the buffer, open will reach the C\$\$IO\$\$ breakpoint. The external agent decodes the CIO buffer, performs an open on this file, and then packs the following response into the CIO buffer:

- length: (int)0
- parmbuf 0-1: dev_fd (2 bytes, little-endian)
- parmbuf 2-7: padding (6 bytes)

dev_fd is the return value of the open function on the host. -1 indicates an error, any non-negative value is a file descriptor.

The external agent then allows the target to resume execution. Upon resuming target execution, open returns a unique file descriptor, which might not have the same value as dev_fd (see below).

File descriptors[\[edit\]](#)

The open function maintains its own notion of file descriptors independently of file descriptors created by any of the devices it manages. open can have more than one device driver, each with their own pool of unrelated file descriptors.

The low-level functions are expected to maintain the mapping of device-file-descriptors to open-file-descriptors. Clients of the low-level functions never see dev_fd.

If CIO is to be used for C standard I/O, open-file-descriptor 0 shall be stdin, 1 shall be stdout, and 2 shall be stderr at program startup without a need for an explicit open in the program.

_DTCLOSE[\[edit\]](#)

close takes argument fd. close will look up the dev_fd corresponding to fd and will format the CIO buffer as a _DTCLOSE system call that will look like this:

- length: (int)0
- command: _DTCLOSE
- parmbuf 0-1: dev_fd (2 bytes, little-endian)
- parmbuf 2-7: padding (6 bytes)

After formatting the buffer, close will reach the C\$\$IO\$\$ breakpoint. The external agent decodes the CIO buffer, performs a close on this file descriptor, and then packs the following response into the CIO buffer:

- length: (int)0
- parmbuf 0-1: result (2 bytes, little-endian)
- parmbuf 2-7: padding (6 bytes)

result is the return value of the close function on the host. -1 indicates an error. 0 indicates success.

The external agent then allows the target to resume execution. Upon resuming target execution, close returns result.

DTREAD[\[edit\]](#)

read takes arguments fd, outbuf, and in_length. If in_length is greater than BUFSIZ, in_length is set to BUFSIZ. read will look up the dev_fd corresponding to fd and will format the CIO buffer as a _DTREAD system call that will look like this

- length: (int)0
- command: _DTREAD
- parmbuf 0-1: dev_fd (2 bytes, little-endian)
- parmbuf 2-3: in_length (2 bytes)
- parmbuf 4-7: padding (4 bytes)

After formatting the buffer, read will reach the C\$\$IO\$\$ breakpoint. The external agent decodes the CIO buffer, performs a read on this file descriptor, and then packs the following response into the CIO buffer:

- length: (int)out_length (or 0 if out_length is -1)
- parmbuf 0-1: out_length (2 bytes, little-endian)
- parmbuf 2-7: padding (6 bytes)
- data: (one ASCII character per byte)

out_length is the return value of the read function on the host. -1 indicates an error. A positive value indicates success; this value may not be more than in_length. out_length may be less than in_length, which is not an error.

The external agent then allows the target to resume execution. Upon resuming target execution, read will copy out_length bytes from _CIOBUF_ at location data into the memory pointed to by outbuf, then return out_length.

DTWRITE[\[edit\]](#)

write takes arguments fd, inbuf, and in_length. write will look up the dev_fd corresponding to fd and will format the CIO buffer as a _DTWRITE system call that will look like this:

- length: (int)in_length in target native endianness
- command: _DTWRITE
- parmbuf 0-1: dev_fd (2 bytes, little-endian)

- parmbuf 2-3: in_length (2 bytes)
- parmbuf 4-7: padding (4 bytes)
- data: (one ASCII character per byte)

After formatting the buffer, write will reach the C\$\$IO\$\$ breakpoint. The external agent decodes the CIO buffer, performs a write on this file descriptor, and then packs the following response into the CIO buffer:

- length: (int)0
- parmbuf 0-1: out_length (2 bytes, little-endian)
- parmbuf 2-7: padding (6 bytes)

out_length is the return value of the write function on the host. -1 indicates an error. A positive value indicates success; this value may not be more than in_length. out_length may be less than in_length, which is not an error.

The external agent then allows the target to resume execution. Upon resuming target execution, write returns out_length.

_DTLSEEK[\[edit\]](#)

lseek takes arguments fd, offset, and origin. lseek will look up the dev_fd corresponding to fd and will format the CIO buffer as a _DTLSEEK system call that will look like this:

- length: (int)0
- command: _DTLSEEK
- parmbuf 0-1: dev_fd (2 bytes, little-endian)
- parmbuf 2-5: offset (4 bytes, little-endian)
- parmbuf 6-7: origin (2 bytes, little-endian)

Possible origins are:

SEEK_SET 0x0000

SEEK_CUR 0x0001

SEEK_END 0x0002

After formatting the buffer, lseek will reach the C\$\$IO\$\$ breakpoint. The external agent decodes the CIO buffer, performs a lseek on this file descriptor, and then packs the following response into the CIO buffer:

- length: (int)0
- parmbuf 0-3: result (4 bytes, little-endian)
- parmbuf 4-7: padding (4 bytes)

result is the return value of the lseek function on the host. (off_t)-1 indicates an error.

The external agent then allows the target to resume execution. Upon resuming target execution, lseek returns result.

DTUNLINK[\[edit\]](#)

unlink takes argument pathname. unlink will format the CIO buffer as a _DTCLOSE system call that will look like this:

- length: (int)0
- command: _DTUNLINK
- parmbuf 0-7: padding (8 bytes)
- pathname: (one ASCII char per byte, including trailing null character)

After formatting the buffer, unlink will reach the C\$\$IO\$\$ breakpoint. The external agent decodes the CIO buffer, performs a unlink on this file descriptor, and then packs the following response into the CIO buffer:

- length: (int)0
- parmbuf 0-1: result (2 bytes, little-endian)
- parmbuf 2-7: padding (6 bytes)

result is the return value of the unlink function on the host. -1 indicates an error.

The external agent then allows the target to resume execution. Upon resuming target execution, unlink returns result.

DTRENAME[\[edit\]](#)

rename takes arguments oldname and newname. rename will format the CIO buffer as a _DTCLOSE system call that will look like this:

- length: (int)strlen(oldname)+strlen(newname)+2 in target native endianness
- command: _DTRENAME
- parmbuf 0-7: padding (8 bytes)
- oldname: (one ASCII char per byte, including trailing null character)
- newname: (one ASCII char per byte, including trailing null character)

After formatting the buffer, rename will reach the C\$\$IO\$\$ breakpoint. The external agent decodes the CIO buffer, performs a rename on this file descriptor, and then packs the following response into the CIO buffer:

- length: (int)0
- parmbuf 0-1: result (2 bytes, little-endian)
- parmbuf 2-7: padding (6 bytes)

result is the return value of the rename function on the host. -1 indicates an error.

The external agent then allows the target to resume execution. Upon resuming target execution, rename returns result.

_DTGETENV[\[edit\]](#)

getenv takes argument varname. getenv will format the CIO buffer as a _DTGETENV system call that will look like this:

- length: (int)strlen(varname)+1 in target native endianness
- command: _DTGETENV
- parmbuf 0-7: padding (8 bytes)
- varname: (one ASCII char per byte, including trailing null character)

After formatting the buffer, getenv will reach the C\$\$IO\$\$ breakpoint. The external agent decodes the CIO buffer, calls the getenv function, and then packs the following response into the CIO buffer:

- length: (int)strlen(data)+1 in target native endianness
- parmbuf 0-7: padding (8 bytes)
- data: (one ASCII char per byte, including trailing null character)

data is the result of getenv("varname") on the host. If getenv returns NULL on the host, a zero-length string is packed into the data instead.

The external agent then allows the target to resume execution. On the target, getenv must arrange for space to hold the returned string, since the environment doesn't exist in target memory. It is unspecified how getenv obtains this memory, which we will call outbuf. Upon resuming target execution, getenv will copy out_length bytes from _CIOBUF_ at location data into outbuf. If the length of data is 0, getenv returns NULL; otherwise it returns a pointer to outbuf. WARNING: Using HOSTgetenv to fetch the value of an environment variable that is longer than BUFSIZ is undefined behavior.

_DTGETTIME[\[edit\]](#)

time takes one argument, t, which is a pointer to time_t. time will format the CIO buffer as a _DTGETTIME system call that will look like this:

- length: (int)0
- command: _DTGETTIME
- parmbuf 0-7: padding (8 bytes)

After formatting the buffer, time will reach the C\$\$IO\$\$ breakpoint. The external agent decodes the CIO buffer, calls the time function, and then packs the following response into the CIO buffer:

- length: (int)0
- parmbuf 0-3: result (4 bytes, little-endian)
- parmbuf 4-7: padding (4 bytes)

result is the number of seconds past the epoch. Note that the TI epoch is NOT the same as the unix epoch. The TI epoch is midnight UTC-6 Jan 1, 1900. See [Time and clock RTS Functions](#). This field is unsigned.

The external agent then allows the target to resume execution. Upon resuming target execution, time assigns result to *t if t is not NULL, then time returns result.

__DTGETTIME64[\[edit\]](#)

__time64 takes one argument, t, which is a pointer to __time64_t. __time64 will format the CIO buffer as a _DTGETTIME system call that will look like this:

- length: (int)0
- command: _DTGETTIME64
- parmbuf 0-7: padding (8 bytes)

After formatting the buffer, __time64 will reach the C\$\$IO\$\$ breakpoint. The external agent decodes the CIO buffer, calls the time function, and then packs the following response into the CIO buffer:

- length: (int)0
- parmbuf 0-7: result (8 bytes, little-endian)

result is the number of seconds past the unix epoch, midnight UTC January 1, 1970. Note that this is a different epoch than _DTGETTIME, above. This field is signed, so it can represent dates before the epoch.

The external agent then allows the target to resume execution. Upon resuming target execution, __time64 assigns result to *t if t is not NULL, then __time64 returns result.

__DTGETCLOCK[\[edit\]](#)

clock takes no arguments. clock will format the CIO buffer as a _DTGETCLOCK system call that will look like this:

- length: (int)0
- command: _DTGETCLOCK
- parmbuf 0-7: padding (8 bytes)

After formatting the buffer, clock will reach the C\$\$IO\$\$ breakpoint. The external agent decodes the CIO buffer. It must then determine, possibly through querying the target's timer registers or querying the simulator, how many CPU cycles have elapsed since the

program began. The return value of clock should monotonically increase; do not make this function return the number of cycles since the last call to clock. WARNING: on very fast devices, this can wrap around. No mechanism is provided to detect this case. The external agent then packs the following response into the CIO buffer:

- length: (int)0
- parmbuf 0-3: result (4 bytes, little-endian)
- parmbuf 4-7: padding (4 bytes)

The external agent then allows the target to resume execution. Upon resuming target execution, clock returns result. The result field is unsigned.

Examples[\[edit\]](#)

C6000 little-endian write[\[edit\]](#)

```
int is 32 bits
char is 8 bits
write(1, "hello\n", 6);
```

target request[\[edit\]](#)

length	0x00	0x06
	0x01	0x00
	0x02	0x00
	0x03	0x00
type	0x04	_DTWRITE
params	0x05	0x01 (might not match fd arg of write, but usually will)
	0x06	0x00
	0x07	0x06
	0x08	0x00
	0x09	don't care
	0x0a	don't care
	0x0b	don't care
	0x0c	don't care
data	0x0d	'h'
	0x0e	'e'
	0x0f	'l'

	0x00	'I'
	0x01	'o'
	0x02	'\n'

external agent response[\[edit\]](#)

length	0x00	0x00
	0x01	0x00
	0x02	0x00
	0x03	0x00
params	0x04	0x06
	0x05	0x00
	0x06	don't care
	0x07	don't care
	0x08	don't care
	0x09	don't care
	0x0a	don't care
	0x0b	don't care

C6000 big-endian write[\[edit\]](#)

```
int is 32 bits
char is 8 bits
write(1, "hello\n", 6);
```

target request[\[edit\]](#)

length	0x00	0x00
	0x01	0x00
	0x02	0x00
	0x03	0x06
type	0x04	_DTWRITE
params	0x05	0x01 (might not match fd arg of write, but usually will)
	0x06	0x00
	0x07	0x06
	0x08	0x00

	0x09	don't care
	0x0a	don't care
	0x0b	don't care
	0x0c	don't care
data	0x0d	'h'
	0x0e	'e'
	0x0f	'l'
	0x00	'l'
	0x01	'o'
	0x02	'\n'

MSP little-endian rename[\[edit\]](#)

```
int is 16 bits
char is 8 bits
rename("abc","def");
```

target request[\[edit\]](#)

length	0x00	0x08
	0x01	0x00
type	0x02	_DTRENAME
params	0x03	don't care
	0x04	don't care
	0x05	don't care
	0x06	don't care
	0x07	don't care
	0x08	don't care
	0x09	don't care
	0x0a	don't care
data	0x0b	'a'
	0x0c	'b'
	0x0d	'c'
	0x0e	'\0'
	0x0f	'd'
	0x10	'e'

	0x11	'f'
	0x12	'\0'

C2000 little-endian rename[\[edit\]](#)

```
int is 16 bits
char is 16 bits
rename("abc","def");
```

target request[\[edit\]](#)

length	0x00	0x08
type	0x01	_DTRENAME
params	0x02	don't care
	0x03	don't care
	0x04	don't care
	0x05	don't care
	0x06	don't care
	0x07	don't care
	0x08	don't care
	0x09	don't care
data	0x0a	'a'
	0x0b	'b'
	0x0c	'c'
	0x0d	'\0'
	0x0e	'd'
	0x0f	'e'
	0x10	'f'
	0x11	'\0'


TI's implementation[\[edit\]](#)

The run-time support library (RTS) shipped with the TI compiler implements the standard C functions fopen, fprintf, etc with calls to the low-level unix-like functions open, close, read, write, lseek, unlink, rename, getenv, gettime, and clock, collectively called the "low-level" functions in the TI compiler user's guide. These functions have the prototypes and semantics you would expect, except that the open function ignores the

mode argument. (This is an artifact; alternate implementations of open may choose to have the unix behavior.) See the compiler user's guide for a description of the hierarchy of the I/O functions in the library.

By default, the low-level functions call the functions HOSTopen, HOSTclose, HOSTread, HOSTwrite, HOSTlseek, HOSTunlink, HOSTrename, HOSTgetenv, HOSTtime, and HOSTclock, collectively called the "HOST device driver functions" in the TI compiler user's guide. These functions implement the CIO interface. These functions funnel all requests through two helper functions, __TI_writemsg and __TI_readmsg. The C\$\$IO\$\$ label is defined at the end of __TI_writemsg.

The C\$\$EXIT label is defined in TI's implementation of abort, which is called at the end of exit and std::terminate.

<div data-bbox="248 1199 852 1323">  <p>Engage in the TI E2E Community Ask questions, share knowledge, explore ideas and help solve problems with fellow engineers</p> </div>	<div data-bbox="852 955 1307 1543"> <p>1. switchcategory:MultiCore=</p> <ul style="list-style-type: none"> For technical support on MultiCore devices, please post your questions in the C6000 MultiCore Forum For questions related to the BIOS MultiCore SDK (MCSDK), please use the BIOS Forum <p>Please post only comments related to the article CIO System Call Protocol here.</p> </div>	<div data-bbox="1307 703 1624 1795"> <p>Keystone=</p> <ul style="list-style-type: none"> For technical support on MultiCore devices, please post your questions in the C6000 MultiCore Forum For questions related to the BIOS MultiCore SDK (MCSDK), please use the BIOS Forum <p>Please post only comments related to the article CIO System Call Protocol here.</p> </div>
---	--	--

Links



[Amplifiers
& Linear](#)
[Audio](#)
[Broadband](#)
[RF/IF &](#)
[Digital](#)
[Radio](#)
[Clocks &](#)
[Timers](#)
[Data](#)
[Converters](#)

[DLP &
MEMS](#)
[High-](#)
[Reliability](#)
[Interface](#)
[Logic](#)
[Power](#)
[Management](#)

[Processors](#)

- [ARM Processors](#)
- [Digital Signal
Processors
\(DSP\)](#)
- [Microcontrollers
\(MCU\)](#)
- [OMAP
Applications
Processors](#)

[Switches &
Multiplexers](#)
[Temperature](#)
[Sensors &](#)
[Control ICs](#)
[Wireless](#)
[Connectivity](#)

Retrieved from

"https://processors.wiki.ti.com/index.php?title=CIO_System_Call_Protocol&oldid=234609"

Category:

- [Compiler](#)

Navigation menu

Personal tools

- [Log in](#)
- [Request account](#)

Namespaces

- [Page](#)
- [Discussion](#)



Variants

Views

- [Read](#)
- [View source](#)
- [View history](#)



More

Search

<input type="text" value="Search"/>	<input type="button" value="Go"/>
-------------------------------------	-----------------------------------

Navigation

- [Main Page](#)
- [All pages](#)
- [All categories](#)
- [Recent changes](#)
- [Random page](#)
- [Help](#)

Toolbox

- [What links here](#)
- [Related changes](#)
- [Special pages](#)
- [Printable version](#)
- [Permanent link](#)
- [Page information](#)

-
- This page was last edited on 8 May 2018, at 12:38.
 - Content is available under [Creative Commons Attribution-ShareAlike](#) unless otherwise noted.

- [Privacy policy](#)
- [About Texas Instruments Wiki](#)
- [Disclaimers](#)
- [Terms of Use](#)

