Introduction

The printf family of functions are standard C functions used to output text. Basic information about the printf family is widely available online (https://en.wikipedia.org/wiki/Printf) and in introductory C textbooks (https://processors.wiki.ti.com/index.php/C_and_C%2B%2B_Language_References). However, users of the TI toolchain need to be aware of certain implementation-specific details.

What is C I/O?

C I/O is a broad term referring to the input and output functions in the RTS; and also to the interface between the low-level RTS functions and the debugger, which allows the debugger to service the user's I/O requests.

All C I/O is funneled through very low-level functions which eventually reach one of two required two breakpoints, at label C$$IO$$ or C$$EXIT. The debugger watches for one of these breakpoints to be reached, at which point the debugger performs an I/O request for the program.

The C standard specifies many input and output functions, and related support functions. This is a partial list.

C standard I/O functions:

- *putc, *getc, *puts, putchar, getchar
- *scanf
- *printf (exception: the *sprintf functions do not require the entire C I/O interface)
- fopen, fclose, fread, fwrite, fflush, freopen
- fseek, fgetpos, fsetpos, ftell, rewind
- perror
- remove, rename
- tmpfile, tmpnam

Other C standard functions which use the C I/O interface:

- getenv
- gmtime, localtime, time, ctime, clock
- exit, abort

The following low-level Unix-style I/O functions are not part of the C standard, but are considered part of the C I/O functions in the RTS:

- open, close, read, write, lseek

Getting C I/O working / Troubleshooting

C I/O is one of the more complicated things in the RTS. It has several layers of abstraction, invokes dynamic memory allocation, has lots of helper functions, and requires intervention from a host debugger to even work at all. For details about the implementation of C I/O, see the Optimizing C Compiler User's Guide section on the C I/O functions.

The typical symptom of not getting everything right is that printf silently fails, and no output is visible in CCS - specifically the "Stdout" window in CCS does not open.

Hello, world!

Start with a simple "hello, world" test to make sure you have configured your system correctly.

```
#include <stdio.h>
int main() { puts("Hello, world!"); }
```

Include `stdio.h`

Every module which uses a C I/O function should include `stdio.h`. Failure to do so can make C I/O functions, particularly `printf`, fail with no warnings generated by the compiler, linker, or debugger. The `printf` function is a variadic function, and calling such a function without a valid prototype in scope invokes undefined behavior. Use the option `--diag_warning=225` to see warnings for all functions used without a valid prototype.

Stack Size

Make sure the stack is large enough. Depending on the version of the compiler, calling one of the *printf functions requires more than 400 bytes of stack. Make sure the stack is in valid read/write data memory.

Heap Size

Most of the standard C I/O functions operate on streams. A stream is stdout, stdin, stderr, or any file opened with fopen. Each stream needs its own I/O buffer, and if you do not provide one, one is automatically dynamically allocated for you when you perform the first read or write. This buffer is of size BUFSIZ (defined in stdio.h) and is by default 256 bytes for all targets.

For projects not using TI-RTOS: Under Project -> Properties -> Build -> Linker -> Basic Options -> Heap Size(-heap) enter the heap size, e.g. 0x400

If using TI-RTOS:

1. Open your tcf file in the configuration tool.
2. Right click on Memory Section Manager and go to Properties.
3. Uncheck the **No Dynamic Heaps** box if it is not already unchecked. Click OK to exit the dialog.
4. Right-click on the memory section where you would like to create a heap, e.g. DDR2 and go to Properties.
5. Click the **Create a heap in this memory** box and enter the size. Click OK to exit the dialog.
6. Right-click once again on Memory Section Manager and set the **segment for malloc/free** to a valid section, e.g. DDR2.

The heap is located in the .sysmem data section. Make sure that the linker command file properly allocates the .sysmem (and .esysmem for C2000) sections to valid read/write data memory.

C I/O Communication Buffer Placement

The .cio data section contains the buffer _CIOBUF_, used by the C I/O functions and the debugger to communicate with each other. You must allocate the .cio section to valid read/write data memory.

Note: This section is rather small, which means that large reads and writes may require many hits of the C$$IO$$ breakpoint to be performed. This buffer can be increased, but the limiting factor is the debugger needs to be able to resize its own internal buffer.

Special Breakpoints

For C I/O to work, the debugger must be able to set breakpoints at the two special labels C$$IO$$ (in function writemsg) and C$$EXIT (in function abort). It's not necessary that these breakpoints be in RAM, but it is necessary that the debugger be able to set these breakpoints. If the bulk of the RTS .text is placed where breakpoints cannot be set, you must separate the input section containing writemsg (trgmsg.obj on most targets, ankmsg.obj on C2000) and abort (exit.obj) and place them into valid read/write program memory.

```
SECTIONS
{
    .text:cio : { rts*.lib<trgmsg.obj exit.obj>(.text) } > RAM
    .text:rts : { rts*.lib(.text) } > ROM
    .text > PMEM
    ...
}
```

Make sure you allow setting of CIO breakpoints at load:

In recent versions of CCS, under Project -> Properties -> Debug -> Program/Memory Load Options make sure the Enable CIO function use is enabled.

Note: The downside of allowing the setting of CIO break-points is that when debugging from Flash, it will use two hardware break-points. You may need to move C I/O code to RAM to avoid this issue.

Flushing

If you set a breakpoint in a program after a C I/O function, you may not see the output taking effect if the stream is buffered. If you need to see the output, you need to make sure the buffer is flushed. You can do this for an _IOLBF stream by making sure to end each printf format string with a '\n' (new line). Since the stdout stream is line buffered this will cause the buffer to be flushed. See the Buffering modes section for further details.

Alternatively, you can flush the buffer manually:

```
fflush(stdout); // This will flush any pending printf output
fflush(FilePointer); // This will flush any pending fprintf output
```

A stream's buffer is automatically flushed when it is closed, and all streams are automatically closed during exit(). If you call abort() (or otherwise prevent the program from reaching exit()) some buffers may not be flushed.

Built-in Limits on the Number of Files

The RTS has a pretty low limit on the number of files you can have open simultaneously. Always check the return value of `fopen()`, `tmpfile()`, and `open()` to confirm the file was opened successfully. If an attempt to open a file fails, check the number of open files. There are two limits on the number of open files.

The first limit is `FOPEN_MAX`, which applies to streams opened with `fopen()`. Streams are represented by a `FILE` pointer. The three standard streams stdout, stdin, and stderr count against this total, so if `FOPEN_MAX` is 10, you may only open 7 more streams without closing one of them. If too many streams are already open, `fopen()` returns `NULL`.

The second limit is `_NSTREAM`, which applies to the low-level C I/O function `open()`. The `open()` function returns an integer representing a file descriptor. If too many file descriptors are already open, `open()` returns -1.

Note that `fopen()` calls `open()` to create a file descriptor for the stream, so each call to `fopen()` counts against the limit for `open()`, as well. Also note that even if the number of files opened with `fopen()` is less than `FOPEN_MAX`, you may not be able to open a new `FILE` if there are too many open file descriptors opened with `open()`.

Finally, note that these limits can be changed by modifying and rebuilding the rts libraries. This forum post (https://e2e.ti.com/support/tools/ccs/f/81/p/7507/29590) provides some details on that process.

Binary Files

If the file you are opening is a binary file, be sure to open the file with a binary mode such as "rb". If you do not, and the file contains data with a '0' byte, the RTS will mistake that for the end of the file. For more see the last paragraph of this discussion (http://c-faq.com/~scs/cclass/int/sx3b.html) on binary data handling.

Avoid C I/O Inside Interrupt Handler Functions

Generally, using any kind of C I/O function from within an interrupt handler is unsafe, because the interrupt may occur during a different call to a function performing C I/O. That is, the C I/O functions are not reentrant.

Reading 8-bit binary data on 16-bit targets

Poses problems. See Reading and Writing Binary Files on Targets With More Than 8-Bit Chars (https://www.ti.com/lit/pdf/spra757)

Performance Considerations

There are many performance considerations when using the C I/O in an embedded system.

Buffered I/O

The C I/O functions are buffered for two reasons:

1. To accumulate smaller I/O requests so that fewer system calls are needed
2. To break up large requests into manageable chunks

Buffering Modes

The C standard provides 3 buffering modes with different performance characteristics:

1. Fully buffered (_IOFBF)
    - The buffer is flushed when it is full.
    - _IOFBF will generate the fewest system calls, and thus provides the best throughput rate
    - You will not see any output at all until the buffer becomes full and is flushed to the output.
    - Files opened with fopen() are _IOFBF.
2. Line buffered (_IOLBF)
    - The buffer is flushed when an end-of-line character ('\n') is encountered.
    - This mode matches the typical expectation of output appearing when a line is printed.
    - The stdout stream is _IOLBF by default.
3. Not buffered (_IONBF)
    - There is no buffering (i.e. each character is output immediately).
    - The stderr stream is _IONBF by default.
    - Each character is output immediately as it arrives, but this may produce an excessive number of system calls, greatly slowing the program

To change the buffering mode for any stream, use the function setvbuf (https://en.wikipedia.org/wiki/Setvbuf). For example:

```
setvbuf(stdout, NULL, _IONBF, 0); // turn off buffering for stdout
```

Dynamic memory allocation in C I/O

Each stream needs its own buffer, and if you do not provide one, one will be automatically dynamically allocated (i.e. malloc) for you when you perform the first read or write. This buffer is of size BUFSIZ (defined in stdio.h) and is by default 256 bytes for all targets. To avoid dynamic allocation by C I/O functions, you must provide a static buffer for each stream you read or write. For example:

```
char static_array[20];
setvbuf(stdout, static_array, _IOLBF, sizeof(static_array));
```

Note: even if you set a buffer to _IONBF, you still have to provide a two-byte static array, or the RTS will allocate a buffer of size BUFSIZ to handle `ungetc()`.

Low-level I/O functions

The low-level I/O functions `read()` and `write()` do not use an I/O buffer. If the request exceeds the size of the _CIOBUF_ buffer, only the first BUFSIZ bytes will be read or written, and the return value will indicate how many bytes were read or written. Thus, if you need to write a large buffer with `write()`, you need to be prepared to place it in a loop.

Stop-mode C I/O

The current C I/O interface works by setting a breakpoint at C$$IO$$. When this breakpoint is hit, the CPU is halted. The debugger then reads the contents of the .cio buffer through JTAG, performs the requested C I/O operation, writes the results back to .cio, and restarts the CPU. This is a very expensive operation in terms of cycles, and can be very disruptive in real-time systems.

Other General Tips

Reduce printf code size with --printf_support

Reduce code size of printf by dropping features ...

- nofloat
- full
- minimal

Additional details are in the article Prinf support in compiler (https://processors.wiki.ti.com/index.php/Printf_support_in_compiler).

Using freopen()

If you have a bunch of calls to `printf()`, but you want the output to go to a file instead of to the standard output, you can reopen stdout with `freopen()`. This will cause all `printf()` output to go to the new file without having to change all of the `printf()` calls to `fprintf()`.

```
if (freopen("output.txt", "w", stdout) == NULL)
    fputs("failed to freopen stdout\n", stderr);
```

Using printf() to output to a user-defined device

You can install a user-defined driver so that you can use the sophisticated buffering of the high-level C I/O functions on an arbitrary device, such as a UART.

The user-defined driver will not use the C I/O interface to communicate with the debugger; the driver will need to provide some other means of output, such as controlling a peripheral.

1. Get/write a device driver for outputting data from the UART (or whatever interface you choose).
2. Write the low-level functions as described in Chapter 8.2 The C I/O Functions of the C Compiler User's Guide.
3. Call `add_device` to add your functions to the stream table (i.e. in addition to stdin, stdout, stderr).
4. Open your stream.
5. Redirect your stream to stdout using `freopen`.
6. Specify what buffering is to be used for your stream by calling `setvbuf`.

For example ...

```
ret_val = add_device("uart", _SSA,
                     uart_open,
             uart_close,
             uart_read,
             uart_write,
             uart_lseek,
             uart_unlink,
             uart_rename);

fid = fopen("uart", "w");
freopen("uart:", "w", stdout);     // redirect stdout to uart
setvbuf(stdout, NULL, _IONBF, 0);  // turn off buffering for stdout
printf("Hello world!\r\n");
```

For additional information, please see the article RAMDISK: A Sample User-Defined C I/O Driver (https://www.ti.com/lit/pdf/spra861).

Additional Tips for TI-RTOS Users

TI-RTOS users should also read more about calling RTS functions from TI-RTOS (https://processors.wiki.ti.com/index.php/DSP_BIOS_FAQ#Q:_Can_runtime_support_.28RTS.29_functions_be_called_when_using_DSP.2FBIOS.3F).

TI-RTOS users have an alternative called `LOG_printf()`. See this FAQ (https://processors.wiki.ti.com/index.php/DSP_BIOS_FAQ#Q:_How_can_I_convert_printf.28.29_statements_to_LOG_printf.28.29.3F) for information on switching from `printf()` to `LOG_printf()`.

Resources

TI Code Generation Tools (https://www.ti.com/tool/TI-CGT)

Related Technical Documents (https://software-dl.ti.com/ccs/esd/documents/ccs_documentation-overview.html)

TI E2E Technical Forums (https://e2e.ti.com)

Last updated: May 21 2022 0:51 Central Time