The `fakedsk` library

Riccardo Bernardini

September 15, 2006

Contents

1 What is `fakedsk`?

`fakedsk` is `C` library which allows you to run a program written for the Texas-Instruments DSK card with a normal PC. It simulates the analog I/O by reading/writing from/to normal files (currently it can read/write WAV and text files). I wrote it to allow my students to test their code in a context which makes easier the debug (you can process simple signals with just a handful of samples) and without having access to the card. My goal was to design something that was simple to use and that allowed for using the *same* source that the students would have been compiled with Code Composer Studio in the lab. This means that this library is especially suited to my owm needs and maybe it is not the most general approach possible.

1.1 What about system requirements?

This version of the library simulates the interrupts coming from the ADC and DAC by using Unix IPC tools (more precisely: process creation, signals, shared memory and semaphores). I tried to be as portable as possible and I believe that my code can be used on every POSIX-compatible system. For sure it worked on Linux and on Windows XP with cygwin+`cygserver` (you need `cygserver` to be able to use the IPC tools). If you are able to use it on other systems, please let me know.

2 How do I use it?

2.1 Requirements

The `fakedsk` library supposes that your program has the structure shown in Fig. 1: you have, besides your own functions,

- an optional interrupt handler for input samples. By default, the interrupt handler name is `c_int12`, but it can be changed (see Section 2.4). In order to read a new sample use the function `input_sample()` which returns a `short` (actually, in `fakedsk` `input_sample()` is a macro)
- an optional interrupt handler for output samples. By default, the interrupt handler name is `c_int11`, but it can be changed (see Section 2.4). In order to write a new sample use the function `output_sample(val)`, where `val` is a `short` (actually, in `fakedsk` `output_sample()` is a macro, so beware of side effects)
- a main program which, after carrying out some initializations, calls the `comm_intr()` function. After the call to `comm_intr()`, the `main()` can enter in an idle loop or do some other work.

2.2 One-shot use

If your program has the required structure and you plan to use `fakedsk` just once or few times, maybe the simplest way to use it is

1. Copy the `*.c` and `*.h` file that you find in the archive in the same directory of your source.
2. `#include`, at the top of your code, the `fakedsk` header with
   ```
   #include "fakedsk.h"
   ```
3. link together with your `C` files the `C` files you find in the library (currently, `fakedsk.c, sampleio.c, semafori.c` and `waveio.c`).

2.3 Regular use

If you plan to use `fakedsk` several times (like I do, to test students' code), it can be more convenient to make a "permanent" installation as follows

1. Extract the archive in a work directory
2. Create the library `libfakedsk.a` with the command `make lib`
3. Move `libfakedsk.a` in a directory searched by your linker
4. Move file `fakedsk.h` in a directory searched by your `C` compiler.

```
/* Uncomment the following line to use fakedsk */
/* #include "fakedsk.h" */

interrupt void
c_int12() // Called when a new input sample is available
{
  /*
   * Use input_sample() to read a new sample, e.g.
   *      short new_sample = input_sample();
   */
 /* Your code... */
}

interrupt void
c_int11() // Called when it is possible to output a new sample
{
 /* Your code... */
  /*
   * Use output_sample() to write a new sample, e.g.
   *
   *      short result;
   *      output_sample(result);
   */
}

main()
{
  /* This code is executed with disabled interrupts */

  comm_intr(); /* Initialize the DSK card and enable the interrupts */

  /* This code is executed with enabled interrupts */
}
```

Figure 1: Example of the program structure necessary to use `fakedsk`

2.4 How do I change the parameters?

`fakedsk` is customizable by `#define`-ing some symbols before `#include`-ing `fakedsk.h`. The `#define`-ble symbols are

`FAKEDSK_INPUT` **(Default: `"input.wav"`)** Name of the file read by `fakedsk`. If the extension is `.txt` or `.TXT`, `fakedsk` supposes that the file is a text file, with a 16-bit signed integer per line, otherwise it suppose that the file is a 16 bit, single channel, PCM coded WAV file.

Please note that some filenames have a special meaning:

- If the filename begins with "`|`" the input (in text format) is read from a pipe opened with `popen()` using as command the filename without the initial `|`. For example, if `FAKEDSK_INPUT` is `"| sine-generator"` `fakedsk` will read the ouput of command `sine-generator`.
- If the filename begins with a "`$`", the filename is taken from the environment variable whose name is the original filename without the initial `$`. For example, if `FAKEDSK_INPUT` is `"$INPUT"` and environment variable `INPUT` is equal to `song.wav`, the input samples will be read from WAV file `song.wav`. This feature allows for changing the input filename without recompiling the program.

  Please note that also filenames read from environment variables are interpreted as described above. Therefore, if the contents of the environment variable begins with a "`|`", a pipe is open, while if it begins with a "`$`", the environment is read again. For example, if `FAKEDSK_INPUT` is `"$INPUT"` and environment variable `INPUT` is equal to `"| sine-generator"`, `fakedsk` will read the ouput of command `sine-generator`; if `INPUT` is equal to `$MIKE` and `MIKE` is equal to `song.txt`, input samples will be read from text file `song.txt`.

`FAKEDSK_OUTPUT` **(Default: `"output.wav"`)** Name of the file written by `fakedsk`. If the extension is `.txt` or `.TXT`, `fakedsk` writes output sample in text format, one sample per line, otherwise it writes a 16 bit, single channel, PCM coded WAV file.

Filenames beginning with "`|`" or "`$`" are interpreted as described for `FAKEDSK_INPUT`.

`FAKEDSK_INPUT_CALLBACK` **(Default: `c_int12`)** Name of the input handler, i.e., the function called when a new sample is ready. If no input handler is defined, `#define` this symbol to `NULL`.

`FAKEDSK_OUTPUT_CALLBACK` **(Default: `c_int11`)** Name of the output handler, i.e., the function called when it is possible to output a new sample. If no output handler is defined, `#define` this symbol to `NULL`.

`FAKEDSK_SAMPLING_FREQ` **(Default: `8000`)** Value, in Hz, of the sampling frequency. `fakedsk` will *try* to call your input/output handler with this

```
#define FAKEDSK_INPUT   "voice.wav"          /* Input from WAV */
#define FAKEDSK_OUTPUT  "result.txt"         /* Output in text format */
#define FAKEDSK_INPUT_CALLBACK   new_sample  /* Input handler name */
#define FAKEDSK_OUTPUT_CALLBACK  NULL        /* No output handler */
#include "fakedsk.h"

interrupt void
new_sample() // Called when a new input sample is available
{
 /* Your code here */
}

main()
{
  comm_intr(); /* Initialize the DSK card and enable the interrupts */

  /* Your code here */
}
```

Figure 2: Example of the customization of the behaviour of `fakedsk`

frequency, but in most cases the real frequency will be much smaller (see Section 2.6 to understand why)

`FAKEDSK_SLOW` **(Default:** ***undefined*****)** If this symbol is `#define`-d to a real value, input/output callback will happen every `FAKEDSK_SLOW` seconds. It is sometimes useful for debugging purposes.

See Fig. 2 for an example.

2.5 What kind of file `fakedsk` reads/writes?

Currently `fakedsk` read and write `WAV` files (single channel, 16 bit, PCM coded) and text file (one sample per line, where each sample is a signed 16 bit integer). The library will distinguish between the two formats on the basis of the filename extension: if the extension is `txt` or `TXT`, the file is considered a text file, otherwise a `WAV` file.

2.6 But my handlers are not called 8000 times per second!

That, unfortunately, is to be expected. `fakedsk` simulates the interrupts from ADC and DAC on the DSK card by launching an *alarm* subprocess which `nanosleep()` for 1/8000 seconds, then wakes up, sends a signal to the main process, and sleeps again. Unfortunately, although `fakedsk` calls `nanosleep()` with the parameter corresponding to 1/8000 s, both the wake up of the *alarm* process and the delivery of signals are handled by the operative system and,

depending on how signals are implemented, it can happen that the maximum resolution is limited by your kernel implementation (for example, on Linux/i386 the maximum time resolution is 10 ms).