STDIO, FILE Pointers, and Serial Ports in Softools C

Softools supports the ANSI FILE structure for stream I/O. This allows use of the stream functions like fread(), fwrite(), fprintf(), fgets(), fscanf() and fputs(). Writing your own STDIO FILE handlers is not that hard and can make the rest of your code easier.

Softools also has stdio, stdin and stdout FILE handles. These can be used or manipulated to redirect stdio to a file, serial port or your own device. This can be done at run-time. DC only supports stdio redirect at compile time.

First, lets look at the FILE structure:

```
typedef int (*__outFunc) (unsigned char, void *);
typedef int (*__inFunc) (void *);
typedef long (*__writeFunc) (const unsigned char far *, long, void *);
typedef long (*__readFunc) (unsigned char far *, long, void *);

typedef struct
  {
  __outFunc             __out;                //      Output routine for
device   */
  __inFunc              __in;                 //      Input routine for
device   */
  __writeFunc           __write;              //      For block devices -
used if not NULL
  __readFunc            __read;               //      Ditto
  int                   __unget;
  //      Must be offset +10 - asm code references this!
  void                  *__userData;   //      Pointer available for any
purpose - never accessed by the LIBs
  }                             FILE;
```

The STDIO libraries use this structure to pass data to and from the device. The __out and __in functions are for single character in/out. The __write and __read are for block transfers. For most devices, the __write and __read functions are NULL and only the character I/O functions are used.

__out sends are character and returns it if successful. If a buffer is full, it returns -1.

__in returns the next character if one is available and -1 if no data is available.

The fread() and fwrite() library functions will use the __read and __write handlers if available. These were added in version 1.68 as calling the device on every character was slower than desired. It was fine for serial, but when FILE pointed to a file system, it got slow due to the overhead. They return the number of characters read or written (may or may not be the same size that was passed to the function.)

__unget() is used by ungetch() to put the last char.

Lastly is __userData. This can be anything you want and can be used to extend FILE with your own data. I used this for the FAT file system, the __userData pointed to a open file in the FAT file system. That way, I could use the stream functions (fprintf() etc) on files with no changes to the libs.

Serial Ports and FILE pointers

There are predefined serial port FILE pointers (defined in rabbit.h):

```
extern FILE SerialA[1];
extern FILE SerialB[1];
extern FILE SerialC[1];
extern FILE SerialD[1];
extern FILE SerialE[1];
extern FILE SerialF[1];
```

It may seem odd they are all arrays of 1 struct. This saves code as you can use the name as an address where a pointer is needed (instead of needing: FILE _SerialA; and then FILE * SerialA=&_SerialA.)

These are FILE * so they can be used in any stream function:

```
fprintf(SerialB,"The current input is %f volts.\r\n",inVolts);
```

That will "printf()" to a serial port directly, no need for sprintf() to a buffer, then send the buffer to the port.

You can also read from a serial port:

```
fgets(SerialB,buff,len);
```

Sounds easy and it is. There is a catch though as explained in the next section.

Serial Ports and overflow/underflow:

Serial ports have buffers defined in the serialInitX() call. What happens when you read from an empty receive buffer or write to a full one? In DC, it was easy, it stopped and waited for data or space. This is very unfriendly to your code as it has a wait loop **outside** your code. Softools is designed with performance as one of its main goals. Rather than wait, it returns back to the caller with the read/write unfinished.

So, if your code is spewing out data using fprintf() faster than the port can send it, you will see data lost. You can prevent this by checking if there is enough room in the transmit buffer before writing (checking SerialSendCountX()).

Similarly, reading via fgetc(SerialX); will return -1 if you read faster than data is available. It is easier to wait until you see the data is there (serialRecvCountX()), then read it all at once.

This may be considered a poor design as the default routines will not work. Softools has a design philosophy that the libs should be minimal and not put arbitrary delays in the code. By not putting an automatic wait in the serial functions, it forces you to implement it the way **you** want, not how someone else has done it. More work, but it is under your control.

Another method with fwrite() is to check the return value. If it matches the number of bytes passed, then all is ok. If less, then calculate the rest that needs to be sent and call again:

Here are some serial write routines that check for a buffer full and retry until done.

```
void serBWriteC(int ch)
{
   while (SerialPutcB(ch)==-1)
      ;   // for CoExec use sleep(10); instead of ';'
 }

void serBWriteS(char * buff,unsigned len)
{
  while (len--)
     serBWriteC(*buff++);
}
```

These will handle writing to a serial port when the buffer gets full. I normally make these short functions 'near' so they can be called faster.

Serial Port FILE Handler with an Automatic Wait.

```
char b_tx_buff[TX_SIZE], b_rx_buff[RX_SIZE];

int near serb_out(unsigned char ch,void * unused)
{
   while (SerialSendCountB()>=TX_SIZE)
     sleep(10); // wait for buffer space available
   if (ch=='\n')
   {
       SerialPutcB('\r');  // change \n to \r\n
       while (SerialSendCountB()>=TX_SIZE)
         sleep(10); // wait for buffer space available
    }
   SerialPutcB(ch);  // sent to tx interrupt routines.
   return ch;
}

int near serb_in(void * unused)
{
   while (SerialRecvCountB()==0)
     sleep(10); // wait for chars
   return SerialGetcB(); // get char from receive interrupt buffer.
}

FILE fSerialB[1]={serb_out,serb_in,NULL,NULL,0,NULL};
```

In your code, you would open the serial port with SerialInitB() and pass the b_tx_buff, and b_rx_buff buffers and sizes. You can then use fSserialB in and stream I/O call:

```
fprintf(fSerialB,"Operation completed in %d seconds.\r",optime);
```

Your output will always go out the serial port and will wait if you write faster than the port is sending.

Notes:

1. The serb_out() handler converts \n to \r\n. This makes writing code easier as you only have to out the \n.
2. The functions call the CoExec task switch function called sleep() with a 10ms wait. If you don't use CoExec, then use your own function or just put a ';' instead of the call.
3. The code assumed TX_SIZE and RX_SIZE are defined somewhere. These are the transmit and receive buffers used by the serial port interrupt routines.

One, final note. If you want to use this code to replace stdio then assign _stdio=fSerialB. Be sure the port is open first.

A Non Interrupt-Driven Serial Console

Let's assume your code is happily outputting debug code to a serial console. It then locks up somewhere. You will likely start looking at the wrong place as there is probably a lot more that went on than shown on the console. When your app locked up, there was probably a lot of messages queued up in the serial buffer that never got out.

Here is a simple non-interrupt driven output FILE handler.

```
#pragma offset_labels on
int near sera_out(unsigned char ch,void * notused)
{
#asm
  ld              a,(ix+.ch)
  cp              '\n'
  jr              nz, .notNL
.sendCR:
  ioi     ld a, (SASR)
  bit     3, a
  jr              nz, .sendCR
  ld              a, '\r'
  ioi     ld (SADR), a
.notNL:
.waitforclear:
  ioi     ld a, (SASR)
  bit     2, a
  jr              nz, .waitforclear
  ld              a,(ix+.ch)
  ioi     ld (SADR), a
#endasm
  return _A;
}

int near sera_in(void * notused)
{
  if ((ini(SASR)&0x80)==0)
     return -1; // -1 if no data available
   return ini(SADR); // read data from port
}
FILE fserial={sera_out,NULL,NULL,NULL,0,NULL};

void InitConsole(void)
{
    if (!_inFlash())
        return;
  SerialSetRate(SER_A,115200l);
  outi(PCFR, srPCFR|=0x40);  // set TXA pin function reg so it is TXA not
bit io.
    _stdio = &fserial;                          //  Direct printf to
SerialA
}
```

This is for serial port A. You will have to change the SADR and SASR to the regs for ports B-F as needed. Also use the appropriate SER_X parameter for SerialSetRate(), and set up the PCFR bits. The _inFlash() check is only for port A as to not interfere with the WinIDE debugger.

Notice it assigns _stdio with &fserial. This redirects the stdio output (printf(), puts() etc) to the serial port. Input is polled also and will return -1 on reads if no char is ready.

The above code is used often in my apps. It is a bit slower as it waits for each char to be transmitted. However, if it locks up, the most you would lose is the last character.

Redirecting Standard I/O

Softools has the standard FILE pointers for stdio. You can override them with your own at run time. That could be used to redirect stdio to a different device.

My network debugger does this. When you connect with the debugger over the Ethernet, it replaces _stdio with debug_stdio. This debug_stdio redirects the stdio streams to the debugger seamlessly to your program. When you close the debugger, it redirects the streams to _stdio.

_stdio is the pointer to the current pointer to a FILE that handles the stdin/stdout I/O streams. You can replace it with your own at any time.

Additional Uses of STDIO and FILE Support

The idea of a stream and a standard library of functions can greatly simplify your code and make it more portable. An example is I have a prototype board with serial Port C while the actual board has serial Port B. In a debug build, I change the FILE structure to handlers for serial C. The production version uses serial port B. The main code just uses the same FILE pointer.

You can also redirect streams as needed. If a FILE is in RAM, you can replace the input and output functions as needed. Suppose you have a serial LCD on one serial port and a keypad done via bit I/O. You could create a FILE with the output handlers for the LCD set to the serial port and input handlers from your own keypad read code:

```
FILE fkbd_lcd=SerialB[0]; // copy serial port B file to our own

int near read_kbd(void * notused)
{
}
```

In your code, reassign fkbd_lcd.__in with read_kbd:

```
 fkbd_lcd.__in=read_kbd;
```

Then you have a single FILE for both of the devices.

Notice the "void *" passed to the I/O functions. If you have a device that needs more functionality, then you can pass a pointer to additional data in the FILE handle.

An alternative, is to define your own struct, with additional fields. This will work fine if you cast the FILE pointer to a FILE when needed. It may break copying structures if the code expects just the default structure.

File systems, printers, displays, and circular log buffers are all examples of things that will benefit from using FILE handles and stdio streams to access them. Your code would be portable and much more readable to new users.

I have extended the FILE handles when using my FAT and EFS file systems. Make the code portable and easy to follow and everyone is familiar with fread(), frwite() and fprintf()-type of functions.

Use with a Real-Time Exec or Cooperative Task Switcher.

If you have a preemptive exec, the you will need to protect the FILE handlers from multiple access with a semaphore. You might want to do this in your application, get semaphore, write a string and then release the semaphore. That way the entire string will be queued atomically. Consider this example:

```
task1()
{
   .....
    fputs(fserialb,"Hello world");
   .....
}

task2()
{
   ...
   fputs(fserialb,"Goodbye Cruel World");
   ...
}
```

where fserialb was a FILE handle redirected to a serial port. Suppose the handler called a task switch if the buffer gets full. Then the next task can run and queue more. So you might get an output like: (colors added for clarity)

```
"Hello Gwooordlbdye Cruel World"
```

As soon as the buffer was full, the OS started alternating tasks as each was waiting for buffer space. In the above example, adding a check/wait for a semaphore before the write and a release afterward will prevent the mixing of multiple tasks writing to the same FILE.

Ok, suppose we have a serial port running at 9600 baud. That is about one character every millisecond. We opened the port with a tx buffer of 1000 chars. That is a full second of data that can be queued. Suppose we have a FILE struct with an __out handler pointing to:

```
int near serb_out(unsigned char ch,void * unused)
{
   while (SerialSendCountB()>=TX_SIZE)
     sleep(1); // wait for buffer space available
   if (ch=='\n')
   {
       SerialPutcB('\r');  // change \n to \r\n
       while (SerialSendCountB()>=TX_SIZE)
         sleep(1); // wait for buffer space available
    }
   SerialPutcB(ch);  // sent to tx interrupt routines.
   return ch;
}
```

Running under CoExec, when the buffer gets full, it will keep retrying every millisecond. Really it is a waste to keep trying so fast. When the buffer is full, we know there is 1 full second of data waiting to go out. So, change the sleep(1) to sleep(100). That will cut down the "churning" down to just a few waits. The buffer will still always be kept between 90-100% full and your other tasks will have more time to run. If you know the baud rate, and buffer size, you can tune the wait to reduce overhead. In the above example the sleep could be 500ms or even 750ms. Longer waits will reduce the number of task switches but will cause longer delays in the writing task. For example with a 500ms wait, if a task wrote 2 characters just as a buffer got filled, it would wait 500ms when it only needed 2ms.

You could use yield; or sleep(0) for minimal latency, but a lot of time will be in task switching. I prefer 10ms as a minimum.

To make writes of strings, use a semaphore:

```
int b_puts(char * buff,unsigned len)
{
  static char serb_sema=0;
    unsigned retry=0;
  while (SEM_LOCK(serb_sema)!=1) // check semaphore
  {
         SEM_UNLOCK(serb_sema); // oops, it is in use, unlock it (dec count
back to 1)
         sleep(10);
        if (++retry==1000) // arbitrary limit of 1000 tries (10 seconds)
           return 0;   // tell caller we tried, but the semaphore was always
busy
  }
    while (len--)
       fputc(fSerialB,*buff++);
  SEM_UNLOCK(serb_sema);
  return 1; // return success
}
```

The above will guarantee that any string will be written to the serial file as one continuous block with no other task data mixed in. It assumes the fSerialB FILE handle has handlers that will take care of buffer overruns (code described in the "Serial Port FILE Handler with an Automatic Wait" section.