Porting Monograph 2 - MIPS EPI/MAJIC

So, you want to run interp on a MIPS target through a E-JTAG emulator?

Even if your target --isn't-- MIPS, this porting example will address common problems you should expect to encounter any time you attempt to port interp to a non-POSIX, emulator or run-at-reset environment, such as memory management and character I/O.

Two directories should have come with this document:
  MIPS-EPI-00.28.00
  MIPS-EPI-SERIAL-00.28.00
The first directory contains the source code changes for the phase I implementation - serial I/O is through the E-JTAG interface. The second dir contains the source code changes for the phase II implementation - serial I/O is through a serial UART internal to the BCM7111.

Because you can obviously read source code, this document will focus on what needed to be done (the "why") instead of wasting too much time telling you "how."

Makefile change:
 1. change the definition of CC from "gcc" to your cross-compiler's
    real name. For example "sb1-elf-gcc"
 2. replace the original Makefile with this one.

 NOTE: There are many other Makefile changes necessary to build a static
       image that runs from a fixed memory location. Those details are in
       the sample Makefile but are not covered to any great depth here.

Environment change:
 1. Know where your cross-compiler executables live? (If not, find them.)
 2. make sure this directory path is in $PATH:
      echo $PATH
 3. If not, add it to the front of your $PATH search string.
 4. For example:
      export PATH=/home/mee/sb1-tools/bin:$PATH

Runtime environment:
The EPI/MAJIC JTAG emulator is bundled with quite a bit of licensed software, that cannot be shown here. The source code that is provided implements a very basic, and primitive runtime environment that superficially resembles the standard (gcc libc) c library, but is actually dramatically reduced.

This is very much standard practice for no-OS, run-at-reset environments, and interp is built specifically to be very adaptable to non-standard libraries with reduced functionality. That is why only the portions of main.c that run on embedded linux use printf(). Everywhere else, I/O is accomplished by calling the interpio and ilowlevelio APIs, of which only ilowlevelio.c requires modification.

Generally, the only other area that might need modification is memory

management. This is encapsulated in the mem API (imem.c.)
Interp has no direct calls to malloc(), calloc(), free(), etc.
As luck would have it, this example requires changes in both areas.

I/O modifications (PHASE I):
The EPI/MAJIC runtime provides a sophisticated mailbox-based mechanism for shuttling serial I/O through the E-JTAG interface. This means that the only physical connection required is the emulator itself.

The I/O modifications are all in one file - ilowlevelio.c. The functions that are affected by porting are:

- poll_input()
- wait_msec()
- get_eof()
- redirected_input()
- map_memory()
- get_char()
- put_char()

Functions not mentioned are generally not affected by porting, and usually because their implementation relies on one or more of the affected functions. One noteworthy function that is not affected by this phase I implementation is init_io(). It will come into play for the phase II implementation (UART) because that interface must be initialized before it can be used. For now, I/O will be handled by the EPI/MAJIC functions (including Boot.S) and the hardware initialization is handled long before interp starts to execute.

poll_input()
This capability is left as an unimplemented stub simply because of the difficulty of providing it in the context of EPI/MAJIC-based I/O. Even so, it is useful to explain how you make this capability into a stub while still allowing existing scripts that use POLL ("p") to mostly work.
If this function always returns true (input is waiting to be read) then the scripts that use it will simply block on input. Although this is not very useful, it implements a more readily understandable behavior than if it always returned false (no input is available) and I/O didn't work at all.

wait_msec()
This is a capability that relies on special-purpose hardware such as a timer or free-running counter. It's not difficult to implement once you have spent the time digging into the manuals and data sheets (that sometimes are "protected" by restrictive or proprietary licenses.) For this tutorial, the function is just an empty stub. The behavior of the WAIT-MSEC ("w") command is that it simply doesn't wait at all. This also isn't very useful, and may even cause mysterious behavior in your existing scripts. If that is the case, then dig into the manuals for your hardware and find one or more registers that let you measure the passage of time (in whatever units) while always being aware that the requested wait time is in milliseconds.

get_eof()
This particular capability is not useful when your environment does not support I/O redirection (or files for that matter.) Specifically, the original implementation relies entirely on a library call that is simply a wrapper for a system call. Or in plain english, even if you can figure out how to implement this, it won't do you any good without some RTOS/OS support for hot-swapping the input (handled by the POSIX dup() and dup2() calls.) This non-functional stub will always return false (no EOF detected.) Your scripts will run normally, and interp won't mind either since it must be built by "make nocli" which removes all the fancy code that deals with I/O redirection - and you don't need to deal with EOF without I/O redirection.

redirected_input()
This stub always returns false (input not redirected), but as mentioned above, there isn't any code that calls it because "make nocli" removes it all.

map_memory()
This function is only required for systems that run interp in protected-mode. Since that isn't the case for emulator-based or RTOS-based environments, this function will always return hardware_address in those enviroments, which makes it functionally a NO-OP. All scripts that use MEMORY-MAP ("Mm") will run normally (if their register addresses and descriptions are correct.)

get_char()
The EPI/MAJIC I/O library prefers block I/O but it can be made to do character I/O. The penalty for doing so is slower performance. Still, it seems fast enough. The one real change here is the substitution of _read() for getchar(). The is no need to echo the character back, or deal with carriage control characters because MONICE (console program that comes with the EPI/MAJIC) handles that itself.

put_char()
Just as with get_char() (above) the only noteworthy change is the substitution of _write() for putchar().

itypes.h
Because the EPI/MAJIC I/O routines want to use small numbers that look like file descriptors (0=stdin, 1=stdout, and 3=stderr), similar definitions were stuck in this file. Except for that, this file would not have any changes for this port because the default CPU architecture is the same as the MIPS - 32-bit machine. If you port to something other than a 32-bit machine, you'll need to figure out the appropriate definition changes to achieve the requirements outlined in the comments in this file.

Memory management alterations:
Typically, primitive memory management routines do not guarantee alignment of any kind. They start on a word boundary (or larger) --but-- allocate EXACTLY the number of bytes requested. The strategy taken here rounds the allocations up to multiples of 4 bytes, and works even with memory management routines that also implement rounding.

The changes here are limited to Mem_alloc(), Mem_calloc(), and Mem_free(). For Mem_alloc() and Mem_calloc(), the assumptions and code changes are identical. The actual allocation routine is assumed to do the first allocation on a word boundary. As long as this is true for you, the code change is simply to round the requested number of bytes up to the next multiple of 4, then call _sysalloc() to perform the allocation.
Mem_calloc() has additional logic after the allocation because _sysalloc() does not guarantee initialization of the storage. It's just a simple loop that sets the storage, if any was allocated, to zero.

Mem_free() calls _sysfree() if the pointer provided is non-NULL.

I/O modifications (PHASE II):
If your target has a serial UART, your should study this additional code change. Most UARTs work generically similar to this one - they have command, data, and status registers. The register sizes (addressing mode) may be different, as well as the bit positions, but the basic ideas remain the same.

For reading, poll it repeatedly until a character appears, read it,

echo the character back to the user's PC, deal with carriage return
and linefeed characters, and return the resulting character.

For writing, poll it repeatedly until the UART is no longer busy
transmitting, write the character, deal with carriage return and return.

The only differences between phase I and II are in put_char(), get_char(),
init_io(), and poll_input(). Also itypes.h does not need any changes from the
baseline version. imem.c is the same for the two versions.

The "#include" files changed a little because the EPI/MAJIC code isn't needed.
Then we need register definitions for the UART CONTROL registers- receiver
status and data, a control register, two baud rate registers, and transmitter
status and data registers.

To keep the changes to put_char() and get_char() very simple, two local
functions, uart_get() and uart_put() were created. uart_get() has a
while loop that could conceivably run forever (no timeout is implemented.)
The expectation is that eventually you will type a character that will
transmit down your serial cable and be picked up by the UART. If that
ever happens, a bit in the receiver status register will say so, and that
is what terminates the wait loop and allows uart_get() to get the newly
arrived character from the receiver data register and return it to get_char().

uart_put() is structured similarly. It will read the transmitter status
register and wait until the UART is finished transmitting the character
from the last time uart_put() was called. Then it places the byte to be
transmitted in the transmitter data register, and returns without waiting
for it to finish transmitting.

init_io() initializes the UART for 8 data bits, no parity, one stop bit (8-N-1)
at 115,200 baud. If you can't use these settings, then refer to the
UART register documentation and pick out settings that you can live with.

get_char() has several modifications. uart_get() replaces _read(). Then
there is additional code to 1) echo the character back across the interface,
and 2) also follow any received RETURN characters with an echoed NEWLINE.
This substitution strategy seems to work no matter what your terminal emulator
transmits (CR/LF or RETURN) for line termination.

put_char() replaces _write() with uart_put() and adds new code that
causes a RETURN (CR) to follow any linefeed (NEWLINE) characters that
are transmitted. Once again this strategy seems to work with all terminal
emulators, since they generally do not expect unix/linux line termination.
They usually expect just a RETURN or a linefeed followed by a RETURN.

poll_input() reads the receiver status register one time and returns
false if no character was received, or true (-1) if a character was received.

Build it for your target:

```
make nocli
```

This is the only way you can sucessfully build interp for this target.

Test it on your target:

Passing script files such as test.int to interp is terminal-emulator
dependent. Refer to your software's documentation. Once you figure this
out, you can send your scripts over to interp (just as if you typed them it)
for execution and see the results scroll up (and off) your display.

You'll also want to figure out how to tell your terminal to log the output it receives from interp to a file (while simultaneously displaying it on the screen.)

Once you have these capabilities figured out you can test interp formally.

You will find differences between golden_results.txt and your captured output. Your differences (as noted in the README) should be limited to steps 08.03, 08.04, 11.05, 12.04, and 18.01. This happens because some of the test results are installation dependent memory addresses.

If your test "failures" are limited just to these test steps, look carefully at the results presented in the difference listing. The results are arranged as pairs of lines:

```
    the actual result
    what the result should be
```

More specifically, the first line is usually a number, and the second line reads "<xx.yy>answer is n". If n is the same as the number on the line above it, then your test actually passed.

You're done!
Yes, that's it. You've successfully ported interp to the EPI/MAJIC execution environment!