Kiwi - a 68k Homebrew Computer

September 7th 2012
The core of the operating system is an adapted version of Enhanced Basic 68k which is a Basic interpreter written in pure 68k assembler. Thanks to its clear design, it is pretty forward to provide custom io routines as well as new commands.
To take advantage of all hardware features which are offered by Kiwi, one wishes to use a higher programming level than offered by assembler or the included basic. This is achieved with an C/C++ cross compiler tool chain. The power of C increases with provided libraries. First to be mentioned is the standard C library *libc*. `printf`, `scanf`, `fopen`, `fwrite`, ... are all defined here. Porting a standard C library means a big leap in software developing. With Newlib Kiwi uses a very portable standard C library.

C/C++ cross compiler tool chain

As the development system runs Linux, it was obvious to build an gcc cross tool chain. When programming assembler, one can define where in the address space the binary code is to be executed (*ORG*) and the addresses at which variables are to be saved. When programming C, one does not need to take care of the position at which the program will be executed or where it will save its variables. To hide the information about the position of the program, the compiler needs to have information about the memory configuration. This is where the *linker script* is for.
It describes the memory layout (size, chunks, stack), as well as where to store different sections as the program code (*.text*), the position where variables are stored (*.bss*) and the static data (*.data*) like "Hello world!" in printf("Hello world!").

```
STARTUP(crt0.o)
OUTPUT_ARCH(m68k)

SEARCH_DIR(.)
GROUP(-lkiwi -lc -lgcc)
__DYNAMIC  =  0;

MEMORY
{
  ram (rwx) : ORIGIN = 0x4214, LENGTH = 0x300000-0x4214
}

/*
 * allocate the stack
 */

/*PROVIDE (__stack = 3M - 8);*/
PROVIDE (__stack = 0);

/*
 * Initialize some symbols to be zero so we can reference them in the
 * crt0 without core dumping. These functions are all optional, but
 * we do this so we can have our crt0 always use them if they exist.
 * This is so BSPs work better when using the crt0 installed with gcc.
 * We have to initialize them twice, so we cover a.out (which prepends
 * an underscore) and coff object file formats.
 */
PROVIDE (hardware_init_hook = 0);
PROVIDE (_hardware_init_hook = 0);
PROVIDE (software_init_hook = 0);
PROVIDE (_software_init_hook = 0);
/*
```

```
 * stick everything in ram (of course)
 */
SECTIONS
{
  .text :
  {
    CREATE_OBJECT_SYMBOLS
    *(.text .text.*)

    . = ALIGN(0x4);
    /* These are for running static constructors and destructors under ELF.  */
    KEEP (*crtbegin.o(.ctors))
    KEEP (*(EXCLUDE_FILE (*crtend.o) .ctors))
    KEEP (*(SORT(.ctors.*)))
    KEEP (*(.ctors))
    KEEP (*crtbegin.o(.dtors))
    KEEP (*(EXCLUDE_FILE (*crtend.o) .dtors))
    KEEP (*(SORT(.dtors.*)))
    KEEP (*(.dtors))

    *(.rodata .rodata.*)

    . = ALIGN(0x4);
    *(.gcc_except_table)

    . = ALIGN(0x4);
    *(.eh_frame)

    . = ALIGN(0x4);
    __INIT_SECTION__ = . ;
    LONG (0x4e560000)   /* linkw %fp,#0 */
    *(.init)
    SHORT (0x4e5e)      /* unlk %fp */
    SHORT (0x4e75)      /* rts */

    . = ALIGN(0x4);
    __FINI_SECTION__ = . ;
    LONG (0x4e560000)   /* linkw %fp,#0 */
    *(.fini)
    SHORT (0x4e5e)      /* unlk %fp */
    SHORT (0x4e75)      /* rts */

    . = ALIGN(0x4);
    _etext = .;
    *(.lit)
  } > ram

  .data :
  {
    *(.got.plt) *(.got)
    *(.shdata)
    *(.data .data.*)
    _edata = .;
  } > ram

  .bss :
  {
    . = ALIGN(0x4);
    __bss_start = . ;
    *(.shbss)
    *(.bss .bss.*)
    *(COMMON)
    _end =  ALIGN (0x8);
    __end = _end;
  } > ram

  .stab 0 (NOLOAD) :
```

```
  {
    *(.stab)
  }

  .stabstr 0 (NOLOAD) :
  {
    *(.stabstr)
  }
```

Every C program contains a function `main` which is to be expected the starting point of the program. But actually, this is not the first code which will be executed. The starting point is the *C runtime library (CRT0)*. The latter is responsible for setting up the correct environment, it deletes variable space (to prevent uninitialized variables having random values), it sets up constructor and destructor lists for C++, if necessary, a new stack is defined, and it is responsible to call `main` with possible arguments. When the program exits, the C runtime library is responsible to clean everything before it finally exits.

```
 /*
 * crt0.S -- startup file for Kiwi
 */

#include "asm.h"

        .title "crt0.S for Kiwi"
#define STACKSIZE       0x8000

/*
 * Define an empty environment.
 */
        .data
        .align 2
SYM (environ):
        .long 0

        .align  2
        .text

/*
 * These symbols are defined in C code, so they need to always be
 * named with SYM because of the difference between object file formats.
 */

/* These are defined in C code. */
        .extern SYM (main)
        .extern SYM (media_start)
        .extern SYM (exit)
        .extern SYM (hardware_init_hook)
        .extern SYM (software_init_hook)
        .extern SYM (atexit)
        .extern SYM(__do_global_dtors)

/*
 * These values are set in the linker script, so they must be
 * explicitly named here without SYM.
 */
        .extern __stack
        .extern __bss_start
        .extern _end

/*
 * Set things up so the application will run. For historical reasons
 * this is called 'start'.  We set things up to provide '_start'
 * as with other systems, but also provide a weak alias called
 * 'start' for compatibility with existing linker scripts.
 */
        .global SYM (start)
        .weak SYM (start)
        .set SYM (start),SYM(_start)

        .global SYM (_start)
SYM (_start):
        /*
         * put any hardware init code here
         */
        move.l  4(sp),a1
        movem.l a3-a6/d0-d7,-(sp)
        /* See if user supplied their own stack (__stack != 0).  If not, then
         * default to using the value of %sp as set by the ROM monitor.
         */
        movel   IMM(__stack), a0
        cmpl    IMM(0), a0
        jbeq    1f
        movel   a0, sp
1:
        /* set up initial stack frame */
        link    a6, IMM(-8)

/*
 * zero out the bss section.
 */
        movel   IMM(__bss_start), d1
        movel   IMM(_end), d0
        cmpl    d0, d1
        jbeq    3f
        movl    d1, a0
        subl    d1, d0
        subql   IMM(1), d0
2:
        clrb    (a0)+
#if !defined(__mcoldfire__)
        dbra    d0, 2b
        clrw    d0
        subql   IMM(1), d0
        jbcc    2b
#else
        subql   IMM(1), d0
        jbpl    2b
#endif

3:

/*
 * initialize target specific stuff. Only execute these
 * functions it they exist.
 */
        PICLEA  SYM (hardware_init_hook), a0
        cmpl    IMM(0),a0
        jbeq    4f
        jsr     (a0)
4:

        PICLEA  SYM (software_init_hook), a0
        cmpl    IMM(0),a0
        jbeq    5f
        jsr     (a0)
5:

/*
 * call the main routine from the application to get it going.
 * main (argc, argv, environ)
 * we pass argv as a pointer to NULL.
 */
```

```
#ifdef ADD_DTORS
         put __do_global_dtors in the atexit list so the destructors get run
        movel   IMM (SYM(__do_global_dtors)),(sp)
        PICCALL SYM (atexit)
#endif
        movel   IMM (__FINI_SECTION__),(sp)
        PICCALL SYM (atexit)

        PICCALL __INIT_SECTION__
/*
        pea     0
        PICPEA  SYM (environ),a0
        pea     sp@(4)
        pea     0 */
        move.l  a1,-(sp)
        PICCALL SYM (media_start)
        move.l  (sp)+,a1
        pea     (a1)
        PICCALL SYM (main) /* Here the main(..) function is called */
        addq.l  #4,sp
        unlk    a6
        movem.l (sp)+,a3-a6/d0-d7
        rts

/*
 * drop down into exit in case the user doesn't. This should drop
 * control back to the ROM monitor, if there is one. This calls the
 * exit() from the C library so the C++ tables get cleaned up right.
 */
        PICCALL SYM (exit)
```

Newlib - portable standard C library

With Newlib, a very portable standard C library is available, where portable means that the hardware depended code is separated from the generic code. For instance, something like `printf` means a lot of the same work for every system. At the very end, `printf` sends one character to the device. This is the only code which has to be written for a new platform. Of course, there are a couple of more functions which rely on the specific hardware, but this is straight forward and Newlib collects all these functions in its *libgloss*. Only a subset of these functions need to be implemented to use Newlib. The more functions are ported, the more functionality is get.

Mixing assembler with C

To use the functions provided by the FAT filesystem library from inside the EhBASIC (as backend for `LOAD` and `SAVE`), it is necessary to call C functions from assembler. Take a look how gcc/m68k handles calling functions with and without arguments and return value. The following code is compiled with

```
m68k-elf-gcc -m68000 -Tkiwi.ld abi.c -o abi
```

```

void function1() {  // NO arguments and NO return value
  return;
}

void function2(int argument) { // ONE argument and no return value
  argument++;
  return;
}
```

```
void function3(int argument1, int argument2) { // TWO arguments and no return value
  int temp;
  temp = argument2;
  argument2 = argument1;
  argument1 = temp;
  return;
}

int function4(int argument) { // ONE argument and return value
  return argument;
}

void main() {
  int retval;
  function1();
  function2(0x1234);
  function3(0x1234,0x5678);
  retval = function4(0x1234);
}
```

The result can be disassembled with objdump:

```
m68k-elf-objdump -d abi
```

```
        ...

000043e8 <function1>:
    43e8:       4e56 0000       linkw %fp,#0
    43ec:       4e71            nop
    43ee:       4e5e            unlk %fp
    43f0:       4e75            rts

000043f2 <function2>:
    43f2:       4e56 0000       linkw %fp,#0
    43f6:       52ae 0008       addql #1,%fp@(8)
    43fa:       4e71            nop
    43fc:       4e5e            unlk %fp
    43fe:       4e75            rts

00004400 <function3>:
    4400:       4e56 fffc       linkw %fp,#-4
    4404:       2d6e 000c fffc  movel %fp@(12),%fp@(-4)
    440a:       2d6e 0008 000c  movel %fp@(8),%fp@(12)
    4410:       2d6e fffc 0008  movel %fp@(-4),%fp@(8)
    4416:       4e71            nop
    4418:       4e5e            unlk %fp
    441a:       4e75            rts

0000441c <function4>:
    441c:       4e56 0000       linkw %fp,#0
    4420:       202e 0008       movel %fp@(8),%d0
    4424:       4e5e            unlk %fp
    4426:       4e75            rts

00004428 <main>:
    4428:       4e56 fffc       linkw %fp,#-4 // getting some space for local variables. Here
                                              // for 4 byte (32 bit integer)

    442c:       4eba ffba       jsr %pc@(43e8 <function1>) // calling the function

    4430:       4878 1234       pea 1234 // pushing argument onto stack
    4434:       4eba ffbc       jsr %pc@(43f2 <function2>) // calling the function
    4438:       588f            addql #4,%sp // add 4 to the stack pointer (actually wiping the
                                             // argument from the stack)
    443a:       4878 5678       pea 5678 // pushing argument onto stack
    443e:       4878 1234       pea 1234 // pushing argument onto stack
    4442:       4eba ffbc       jsr %pc@(4400 <function3>) // calling the function
    4446:       508f            addql #8,%sp // add 8 to the stack pointer (actually wiping the
                                             // arguments from the stack)


    4448:       4878 1234       pea 1234 // pushing argument onto stack
    444c:       4eba ffce       jsr %pc@(441c <function4>) // calling the function
    4450:       588f            addql #4,%sp // add 4 to the stack pointer (actually wiping the
                                             // argument from the stack)
    4452:       2d40 fffc       movel %d0,%fp@(-4) // store return value

    4456:       4e5e            unlk %fp
    4458:       4e75            rts
        ...
```

Every function starts with an `linkw` instruction and ends with an `unlk` followed by an `rts` instruction. The `linkw` instruction allocates some space for local variables. For instance, `linkw %fp,#-4` allocates space for four bytes (for the 32 bit integer variable `retval`) from the stack. If a function does not use local variables, the allocated space is 0. This is useless and unnecessary overhead. These unnecessary frame pointer memory allocations can be avoided by adding the option `-fomit-frame-pointer` to gcc at compile time.
Let us examine each C-function and their produced assembler code.

- function1: Without arguments or return value, calling this function is done with a single *jsr*.
- function2: Here `pea 1234` pushes 0x1234 (the argument) onto stack before jumping to the function. After returning, the stack pointer is incremented by four to wipe the argument from the stack.
- function3: There are two `PEA` instructions before the function is called, whereby there is one for both arguments. The `LINK` instruction inside the function allocates four bytes for the local integer variable temp. After returning, the stack pointer is incremented by eight (2 arguments, 4 bytes each) to wipe the arguments from the stack.
- function4: The calling is like in function2 but with an return value. After returning from the function, the return value is available in register D0. Again, the argument is wiped from the stack by incrementing its pointer by four.

With these information it is easy to write assembler code which calls C functions. You push every single argument to the stack starting from the last, call the function as subroutine, and expect the return value in register D0. Don't forget to wipe all arguments from stack afterwards.

ROM and booting system

The 32KB Eprom could store the plain basic interpreter. To speed up the development, a simple boot code has been written. After power-on it sets up the stack and tries to load and run KERNEL.SYS from harddisk. KERNEL.SYS could be any executable located to run at address 0x3e8000. When using C, the linker script has to be modified to achieve this. Kiwi uses the binary of the Basic interpreter as KERNEL.SYS.

Basic interpreter

The complete basic interpreter consists of the core, which is written in pure assembler, and the FAT filesystem module, which is written in C. The source code of the Basic interpreter is written in an syntax, which is is not compatible with the GNU GCC assembler. It can be assembled with the unix68k assembler though. To avoid porting the source code to GCC assembler syntax, I built both modules with their native tools and binded them afterwards. The calling scheme is not pretty yet, since the calls are just built around absolute jumps. It would be more elegant to install trap vectors and just call them. For now, to build a complete basic interpreter it takes two manual passes:

- Assemble the core of the basic interpreter. Note, where the binary ends in address space.
- Modify GCC's linker script to compile the code immediately after the end of the core.
- Compile the FAT filesystem library.
- Use *objdump* to find the addresses of all needed functions and copy them to the defines of the core's assembler source.
- Assemble the core again (this time with the correct defines) and copy the filesystem binary immediately after it.

You need to take care of the word boundaries. In some cases, you have to add a fill byte to the end of the cores binary to make sure the filesystem binary starts at an even address.

FAT 12/16/32 filesystem

Originally, the filesystem library has been designed for a small memory footprint. This involves reading single (FAT) sectors at a time. For 16 and 32 bit FAT entries, this is just fine. As with a 512 byte sector, every FAT table entry stays within sector boundaries. But this does not hold anymore for 12 bit FAT table entries, where every entry takes one and a half byte. *512/1.5=341.333, 1025/1.5=682.666,* ***1536/1.5=1024*** This is maybe the reason why the library did not support FAT12 originally.
To simplify adding FAT12 (to support floppy disks), I changed the library such that it always reads three consecutive FAT sectors at a time.

TCP/IP protocol stack

lwIP provides three APIs for the application development. Only the raw API is used, since the two higher APIs do need support from a multitasking operating system. To implement all the needed functionality, the raw API makes use of callback functions and the system timer. A simple framework to get the stack up and running could look like the following code.

```
#include "lwip/inet.h"
#include "lwip/tcp.h"
#include "lwip/ip_frag.h"
#include "lwip/netif.h"
#include "lwip/init.h"
#include "lwip/stats.h"
#include "netif/etharp.h"
#include "../../TCP/kiwi/include/netif/cs8900if.h"
#include "echo.h"
#include <stdio.h>
#include <fcntl.h>

#define PIT_BASE        0x003dfb00
#define PSR             0x1a
#define TCR             0x20
#define CPRH            0x26
#define CPRM            0x28
#define CPRL            0x2A
#define CRH             0x2e
#define CRM             0x30
```

```
#define CRL             0x32
#define TSR             0x34
#define READ_PIT(x)     (*((volatile char *) PIT_BASE + (x) ))
#define WRITE_PIT(x, y) (*((char *) PIT_BASE + (x) ) = (y) )

struct netif netif;

unsigned long frame=0;
void ISR7(void) {
        asm volatile ("movem.l   %a0-%a6/%d0-%d7,-(%sp)");      // Save register values onto sta
        frame+=10;
        WRITE_PIT(TSR,0x1);
        asm volatile ("movem.l   (%sp)+,%a0-%a6/%d0-%d7");      // Get back register values
        asm volatile ("rte");   // Return from Exception
}

void init_timer() {
        WRITE_PIT(TCR,0xe0); // Stop timer
        WRITE_PIT(CPRH,0x01);   // Preload timer (high,medium,low) 10000
        WRITE_PIT(CPRM,0x86);
        WRITE_PIT(CPRL,0xa0);
        WRITE_PIT(TCR,0xe1); // Start timer
}

unsigned long sys_now() {
        return frame;
}

void main() {
        asm volatile ("move.w   #0x2700, %sr"); // Disable interrupts
        (*((long *)0x8) = (long)ISR1);  // Bus error
        (*((long *)0xc) = (long)ISR2);  // Address error
        (*((long *)0x10) = (long)ISR3); // Illegal instruction
        (*((long *)0x14) = (long)ISR4); // Zero divide
        (*((long *)0x18) = (long)ISR5); // CHK instruction
        (*((long *)0x20) = (long)ISR6); // Privilege violation

        (*((long *)0x64) = (long)ISR7); // Level 1 interrupt autovector
        (*((long *)0x74) = (long)ISR7); // Level 5 interrupt autovector
        #define PACKETPP *((volatile u16_t *)(0x3dff00 + 0x0A))
        #define PPDATA   *((volatile u16_t *)(0x3dff00 + 0x0C))

        /* Network interface variables */
        struct ip_addr ipaddr, netmask, gw;

        /* Set network address variables */
        IP4_ADDR(&gw, 192,128,2,1);
        IP4_ADDR(&ipaddr, 192,168,2,50);
        IP4_ADDR(&netmask, 255,255,255,0);

        /* The lwIP single-threaded core: initialize the network stack */
        lwip_init();

        /* Bring up the network interface */
        netif_add(&netif, &ipaddr, &netmask, &gw, NULL, cs8900if_init, ethernet_input);
        netif_set_default(&netif);
        netif_set_up(&netif);

        init_timer();
        asm volatile ("move.w   #0x2000, %sr"); // Enable interrupts
        etharp_init();
        echo_init();
        int flags;
        while (1) {
                sys_check_timeouts();
                cs8900if_service(&netif);
                sendkey();
        }
}
```