Application Note

System-on-Chip Lite+

Development Board

µC Linux-Kernel 2.4.24 and Application Software Information

Internal Document No. TPS-HE-A-1046
Date Published   August 2005 N

NOTES FOR CMOS DEVICES

① PRECAUTION AGAINST ESD FOR SEMICONDUCTORS

Note:

Strong electric field, when exposed to a MOS device, can cause destruction of the gate oxide and ultimately degrade the device operation. Steps must be taken to stop generation of static electricity as much as possible, and quickly dissipate it once, when it has occurred. Environmental control must be adequate. When it is dry, humidifier should be used. It is recommended to avoid using insulators that easily build static electricity. Semiconductor devices must be stored and transported in an anti-static container, static shielding bag or conductive material. All test and measurement tools including work bench and floor should be grounded. The operator should be grounded using wrist strap. Semiconductor devices must not be touched with bare hands. Similar precautions need to be taken for PW boards with semiconductor devices on it.

② HANDLING OF UNUSED INPUT PINS FOR CMOS

Note:

No connection for CMOS device inputs can be cause of malfunction. If no connection is provided to the input pins, it is possible that an internal input level may be generated due to noise, etc., hence causing malfunction. CMOS devices behave differently than Bipolar or NMOS devices. Input levels of CMOS devices must be fixed high or low by using a pull-up or pull-down circuitry. Each unused pin should be connected to V_{DD} or GND with a resistor, if it is considered to have a possibility of being an output pin. All handling related to the unused pins must be judged device by device and related specifications governing the devices.

③ STATUS BEFORE INITIALIZATION OF MOS DEVICES

Note:

Power-on does not necessarily define initial status of MOS device. Production process of MOS does not define the initial operation status of the device. Immediately after the power source is turned ON, the devices with reset function have not yet been initialized. Hence, power-on does not guarantee out-pin levels, I/O settings or contents of registers. Device is not initialized until the reset signal is received. Reset operation must be executed immediately after power-on for devices having reset function.

Regional Information

Some information contained in this document may vary from country to country. Before using any NEC product in your application, please contact the NEC office in your country to obtain a list of authorized representatives and distributors. They will verify:

- Device availability
- Ordering information
- Product release schedule
- Availability of related technical literature
- Development environment specifications (for example, specifications for third-party tools and components, host computers, power plugs, AC supply voltages, and so forth)
- Network requirements

In addition, trademarks, registered trademarks, export restrictions, and other legal issues may also vary from country to country.

NEC Electronics America Inc.
Santa Clara, California
Tel: 408-588-6000
800-366-9782
Fax: 408-588-6130
800-729-9288

NEC Electronics (Europe) GmbH
Duesseldorf, Germany
Tel: 0211-65 03 1101
Fax: 0211-65 03 1327

Sucursal en España
Madrid, Spain
Tel: 091- 504 27 87
Fax: 091- 504 28 60

Succursale Française
Vélizy-Villacoublay, France
Tel: 01-30-67 58 00
Fax: 01-30-67 58 99

Filiale Italiana
Milano, Italy
Tel: 02-66 75 41
Fax: 02-66 75 42 99

Branch The Netherlands
Eindhoven, The Netherlands
Tel: 040-244 58 45
Fax: 040-244 45 80

Branch Sweden
Taeby, Sweden
Tel: 08-63 80 820
Fax: 08-63 80 388

United Kingdom Branch
Milton Keynes, UK
Tel: 01908-691-133
Fax: 01908-670-290

NEC Electronics Hong Kong Ltd.
Hong Kong
Tel: 2886-9318
Fax: 2886-9022/9044

NEC Electronics Hong Kong Ltd.
Seoul Branch
Seoul, Korea
Tel: 02-528-0303
Fax: 02-528-4411

NEC Electronics Singapore Pte. Ltd.
Singapore
Tel: 65-6253-8311
Fax: 65-6250-3583

NEC Electronics Taiwan Ltd.
Taipei, Taiwan
Tel: 02-2719-2377
Fax: 02-2719-5951

Table of Contents

List of Figures

1. Quick Start

1. You will need a computer running a version of Intel- resp. i386-Linux, with sufficient RAM and harddisk space. Make sure a GNU C compiler toolchain is installed. Due to the multitude of Linux distributions available, we cannot give precise information what packages to install. But if your installer or setup program offers a predefined setting named like "Software Development", that might be a good starting point. To use the GUI configuration (`make xconfig`), you also need a Tcl/Tk package installed (it's probably part of most standard configurations). Approximately 1.5 GB additional disk space is needed for the cross-toolchain, uClinux-dist and the temporary files created during the build.
2. You will need a cross-compiler package for your target. Many binary tool packages exist specifically for compiling uClinux. As you are targeting ARM systems then you can use the arm-elf-tools binary packages of `www.uclinux.org`, which are included on the NEC-µCLinux-package. To install it, change to super-user mode (`su` command). Then change to the NEC-µCLinux-package directory and type in

   ```
   sh arm-elf-tools-20030314.sh
   ```

   For all further steps, you should not be logged in as root. Developing as root is not recommended at all. So leave the super-user mode by the command `exit`.
   If `/usr/local/bin` is not in your shell's program search path, you should add it (example for the bash shell):

   ```
   PATH=$PATH:/usr/local/bin
   export PATH
   ```

3. If you have not un-archived the source package then do that now. You can do this into any directory; typically use your own user home directory. So, change to your home directory, then type:

   ```
   tar xvzf <NEC-uCLinux-package path>/uClinux-dist.tar.gz
   ```

   This will dump the source into an uClinux-dist directory.

4. Change into the source tree:

   ```
   cd uClinux-dist
   ```

5. Configure the build target:

   ```
   make xconfig
   ```

   You can also use `make config` or `make menuconfig` if you prefer. The top level selection is straight forward if you know the vendor of the board you want to compile for. You can also choose to modify the underlying default kernel and application configuration if you want (Figure 1-1).

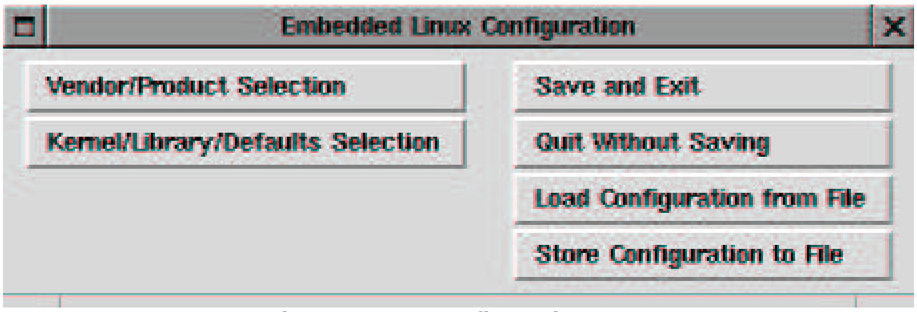

Figure 1-1: Xconfig main

At first it is suggested that you use the default configuration for your target board. It will almost certainly work "as is". For the System-on-Chip Lite+ board, you should select vendor "NEC" and product "SOClite+" in the Vendor/Product selection dialog (Figure 1-2).

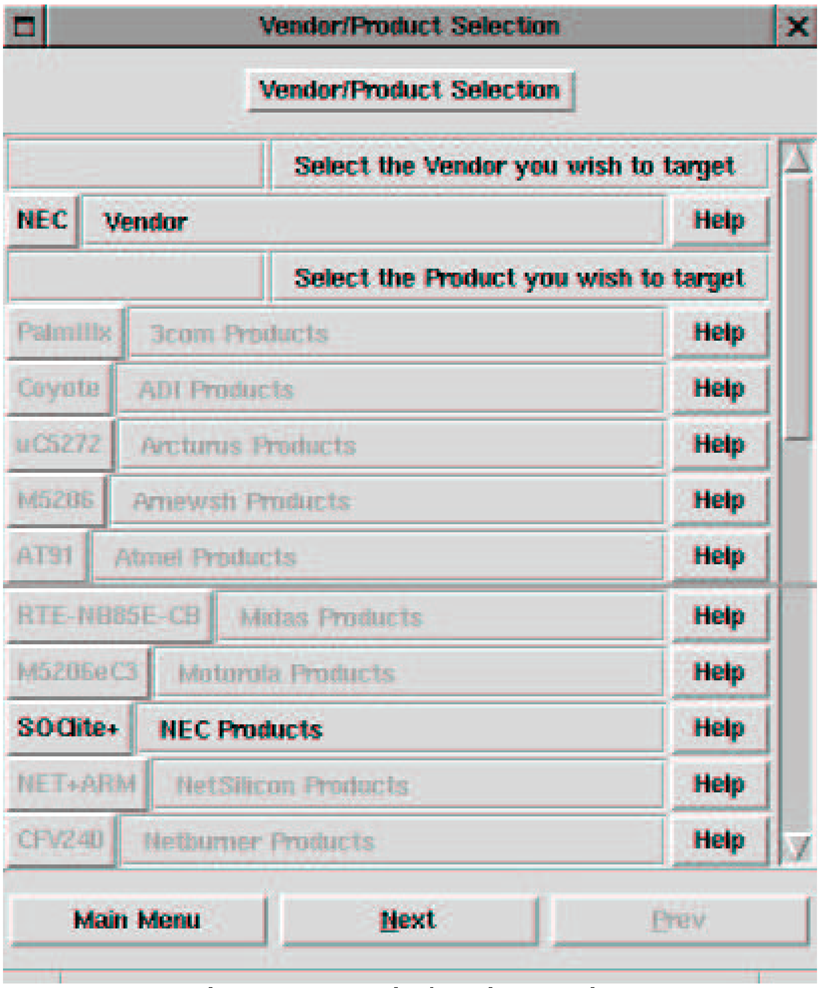

Figure 1-2: Vendor/Product setting

In the Kernel/Library selection box you should choose kernel "2.4.x" and library "uClibc" (other versions are not supported) and check "Default all settings" (Figure 1-3).

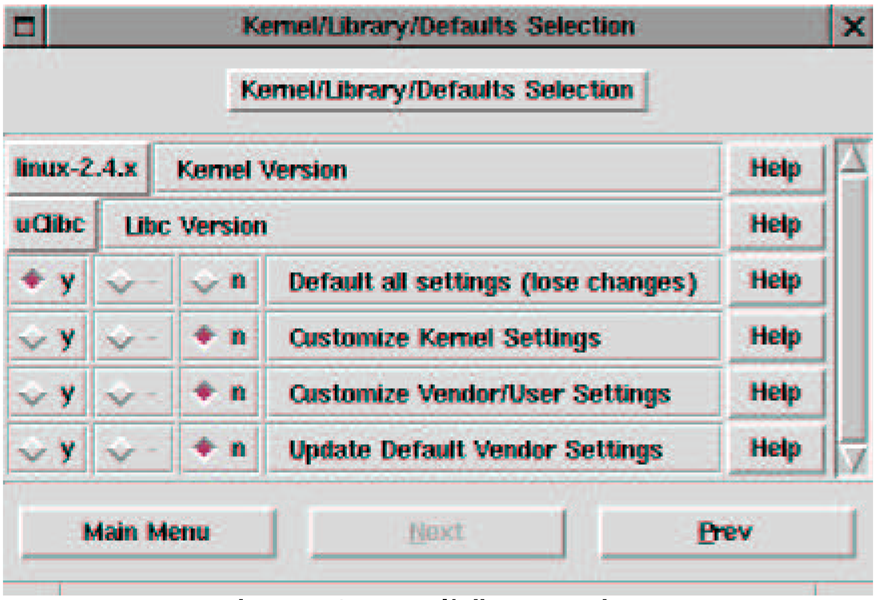

Figure 1-3: Kernel/Library settings

Click <Main Menu> and <Save and Exit>

6. Build the dependencies:

```
make dep
```

7. Build the image:

```
make
```

That's it!
The make scripts will generate appropriate binary images for the target hardware specified. The build process can take some time (15 minutes on a 2 GHz Pentium). Approximately 1.3 GB disk space are needed for the installed toolchain, uClinux-dist and the temporary files created during the build. All generated target files will be placed in the images directory. The exact files vary from target to target, for the System-on-Chip Lite+ board you end up with three binary images, each available in raw binary form and as Motorola S-record file:

- `kernel.bin, kernel.srec`
  The Linux kernel as a binary image, without any debug information. This image could be programmed into the flash.

- `jffs2.img, jffs2.srec`
  The root file system in JFFS2 format. JFFS2 (Journalling Flash File System) is a compressing, rewritable file system especially for flash memory.

- `romfs.img, romfs.srec`
  The same root file system in ROMFS format. ROMFS is a read-only, not compressing file system, but it is faster than JFFS2.

8. Load the image(s) into the target

Make sure all the jumpers and DIP switches are in their default positions. Switch SW2-3 is of special interest here, because it selects if the processor boots from internal ROM (On position) or from external Flash memory (Off position) (Figure 1-4).

Jumpers JP20 (ANEG), JP21 (DPLX) and JP22 (Speed) select the network connection type and speed. By default, the board is configured to a full-duplex 10Mbit connection. If you experience connection problems, you can use these jumpers to set alternative connection parameters or auto-negotiate. In Figure 1-4, a half duplex, 10 Mbit connection is selected.

How to load and run the generated images will depend on your target system and your debugging hardware.

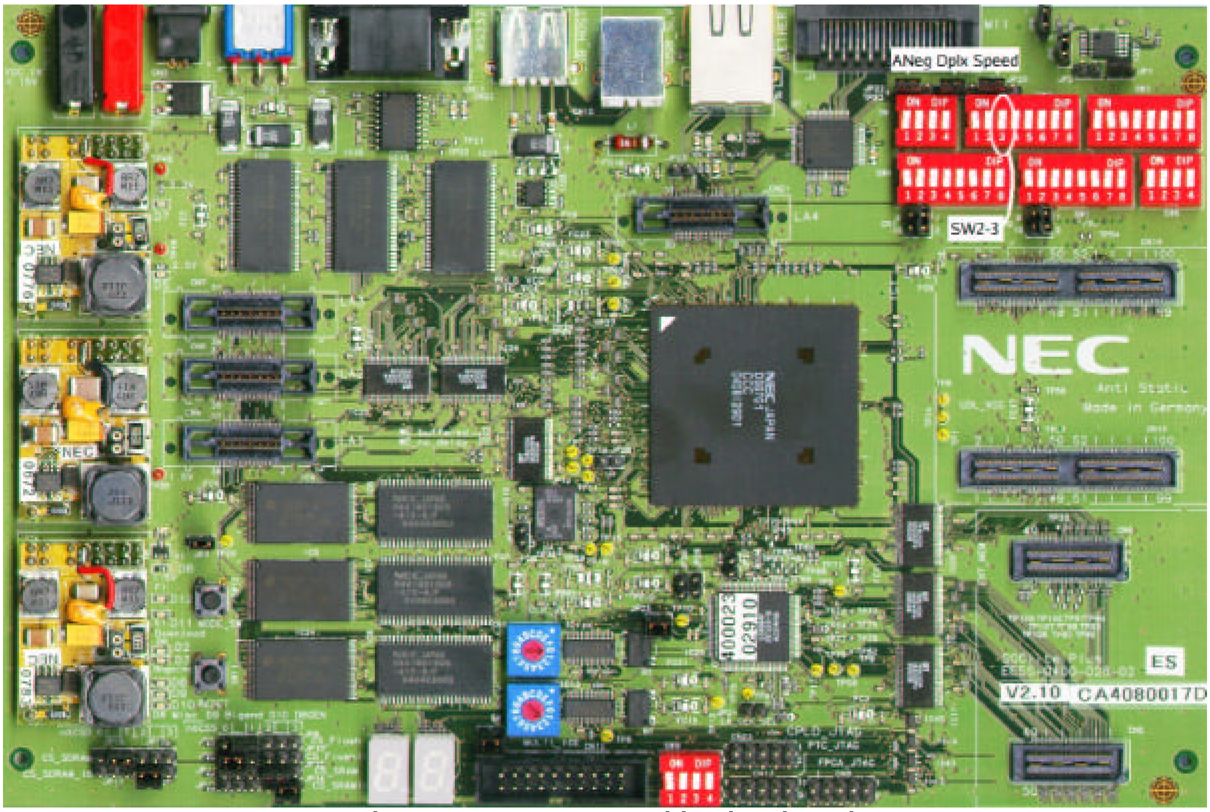

Figure 1-4: System-on-Chip Lite+ board

If you want to download the images over the serial interface, you can use the flash load tools supplied by NEC. You also need a terminal program that can send ASCII files, like *minicom*, *kermit* or *Teraterm*.

The Flash erase and load programs can be found on the NEC-uCLinux-package in the /FlashLoad directory, while the Linux kernel and file system images can be found in the /uCLinux-dist /images directory

- Set your terminal programm to 57600 baud, 8N1
- Set switch SW2-3 to the ON position (Bootloader mode)
- Erase the Flash
    - Reset board
    - Press '>' to synchronize the UART. The Welcome message will appear.
    - Press 's' and <Return> to start the download receive function
    - Send the s-record Erase_AMD_Chip.m32 as ASCII file. Wait until the transmission is finished.
    - Type "g 8000000" to execute the flash program.
    - Press '>' to synchronize the UART again. The message “erasing” is displayed. The erase procedure requires several minutes!!! Please wait for the final checks, displaying ”Verify” followed by several “Address-Data” outputs!!! After the last address 007F0000 you may continue.

- Program the kernel
  - Reset board
  - Press '>' to synchronize the UART. The Welcome message will appear.
  - Press 's' and <Return> to start the download receive function
  - Send the s-record Flash_Load_AMD.m32 as ASCII file. Wait until the transmission is finished.
  - Type "g 8000000" to execute the flash program.
  - Press '>' to synchronize the UART again. A download message appears.
  - Send the kernel s-record kernel.srec as ASCII file (No other input is allowed before sending the file!).
    When the transmission is finished, the necessary flash sectors will be programmed.

- Program the JFFS2 file system
  - Reset board
  - Press '>' to synchronize the UART. The Welcome message will appear.
  - Press 's' and <Return> to start the download receive function
  - Send the s-record Flash_Load_AMD.m32 as ASCII file. Wait until the transmission is finished.
  - Type "g 8000000" to execute the flash program.
  - Press '>' to synchronize the UART again. A download message will appear.
  - Send the JFFS2 s-record jffs2.srec as ASCII file (No other input is allowed before sending the file!).
    When the transmission is finished, the necessary flash sectors will be programmed.

If you want to download the kernel with a debugger, you will find the kernel in ELF format (with debug information) under linux-2.4.x/linux. A hardware JTAG interface provides a quick and easy way to load data on your target system. We found that EPI's JEENI [6] works well with the GNU debuggers *gdb* and *insight*.

For the initial gdb download, you will need a kernel with a builtin file system. To generate one of your own, in the kernel configuration (Figure 2.1) following steps need to be done:

Make sure that you are working in the `uCLinux-dist` directory and type in:

```
make xconfig
```

You will see the dialog box shown in Figure 1-1. Now proceed as follows:

- press “Load Configuration from File” and enter the following file name:
  `vendors/NEC/SOClite+/config.linux-2.4.x-romfs-builtin`
- press “OK”.
- Make shure that Vendor/Product Selection and Kernel/Library/Defaults Selection are defined as described in step 5.
- Press “Save and Exit”

To build the image, type:

```
make dep
make
make
```

As the ROMFS file system integrated into the kernel is the one generated by the previous make run, you have to call make twice (compared with step 7) to make sure kernel and applications are up to date.

Don't try to program this "builtin-FS" kernel into the flash memory; it won't fit into the kernel partition.

Start the debugger (`insight-5.2.1/gdb/gdb`), execute the `.gdbinit` script to initialize the target board hardware, download and run the kernel. In the GDB/Insight console window, it would look like this:

```
(gdb) file binary-images/kernel-builtin-romfs.elf
(gdb) source uClinux-dist/linux-2.4.x/.gdbinit
(gdb) load
(gdb) cont
```

For the next steps, we need the "default-configuration" kernel without builtin FS image. We have to transfer the kernel and JFFS2 images to System-on-Chip Lite+'s ramdisk using FTP or NFS, and program them into the Flash memory using *mtdw* (section 4.6.1). If your host computer has for example the IP address 192.168.30.254, is running an NFS server, and exports your development directory `/LinuxSOC`, you can execute these steps from your target system's console like this:

```
#  mount -t nfs -o  nolock 192.168.30.254:/LinuxSOC /mnt
#  cd /var
#  cp /mnt/uClinux-dist/images/jffs2.img
#  mtdw -e jffs2.img /dev/mtd1
#  cp /mnt/uClinux-dist/images/kernel.bin
#  mtdw kernel.bin /dev/mtd0
```

Start Linux
Set switch SW2-3 to the OFF position (Flash boot mode) and reset the board. Linux should start.

9. Set the Ethernet MAC and IP address

The System-on-Chip-Lite+ board has an on-board Ethernet controller, for which a MAC address and an Ethernet IP address have to be set in order to avoid network collisions.

- Set-up the Ethernet IP address

  A valid Ethernet IP address must be provided by your network administrator together with the sub-net mask and gateway address.

  For temporary testing, the address can be set on run-time using the ifconfig command (ifconfig eth0 add <ip-address> netmask <net mask>), however is lost after reset.

  A permanent change should be made, after µCLinux is running (after Step 9). See also chapter 9.3.

- Set-up the Ethernet MAC address
  For the on-board Ethernet controller a MAC address has been reserved in order to avoid network collisions. The MAC addresses are generally given as:

  00:00:4C:80:5A:nn

  "nn" has to be calculated from your board's serial number, that can be found on a sticker on the board itself.

  - for serial numbers CA4080011D to CA4080015D:
    nn = last 3 numbers of the serial number converted to hex + 70h
  - for other serial numbers:
    nn = last 3 numbers of the serial number converted to hex + 80h

Note: Please note, that the MAC address is counted up as a hexadecimal number while the NEC serial number is counted up in decimal.

Here are examples:

NEC serial number	CA4080013D	first	MAC address:	00:00:4C:80:5A:7D
NEC serial number	CA4080020D	first	MAC address:	00:00:4C:80:5A:94

The 3 higher bytes of the MAC address are defined at Kernel compile time and need not be changed. So nothing to do for them here. Please refer to the chapter 3.2.2.

The lower 3 bytes have to be configured **BEFORE** connecting the board to the Ethernet network. These 3 Bytes are implicit part of an 8-Byte serial number, which µCLinux allows to write into the configuration area.

This µCLinux port uses the JFFS2 file system, and the configuration area is on the rom-device /dev/mtd2.

The program "ksetup" (See chapter 4.6.2) is used to write the data.

For changing the MAC address of the board use following procedure:

- Enter command "ksetup /dev/mtd2"
- The the kernel command line is requested. The default need not be changed, so press return.
- The higher serial number part must be entered. Before entering the new value the software provides a default value. This number is not fixed by NEC. If it shall not be changed, simply press Return.
- The lower serial number part must be entered. The lower 3 bytes equal the lower 3 bytes of the Ethernet MAC serial number. Enter e.g. 0x00805Ann, with nn= calculated number from the serial number of the board.
- After return the data is written to the Flash.
- As the data is read by the kernel on start-up, the data becomes valid after the next µCLinux restart. The current MAC address may then be checked with the command ifconfig.

[MEMO]

2. Configuring uClinux-dist

This distribution is based on the package *uClinux-dist*. Besides the source code for kernel, libraries and applications, *uClinux-dist* also contains configuration scripts and a set of make files to build the complete target system software in one go.

In general, you can select between different kernel versions and libraries, at the top level. Not all kernel versions support all boards, as a general rule choose 2.4.x. Also typically you would use glibc only on target processors that support virtual memory (x86, SH4, XSCALE). Most MMUless processors use uClibc. For the System-on-Chip Lite+ board always choose kernel 2.4.x and uClibc.

Based on what platform you choose in this step the build will generate an appropriate default application set.[1]

To set your own configuration, select "Customize Kernel Settings" or "Customize Vendor/User Settings" in the "Kernel/Library" dialog (Figure 1-3).

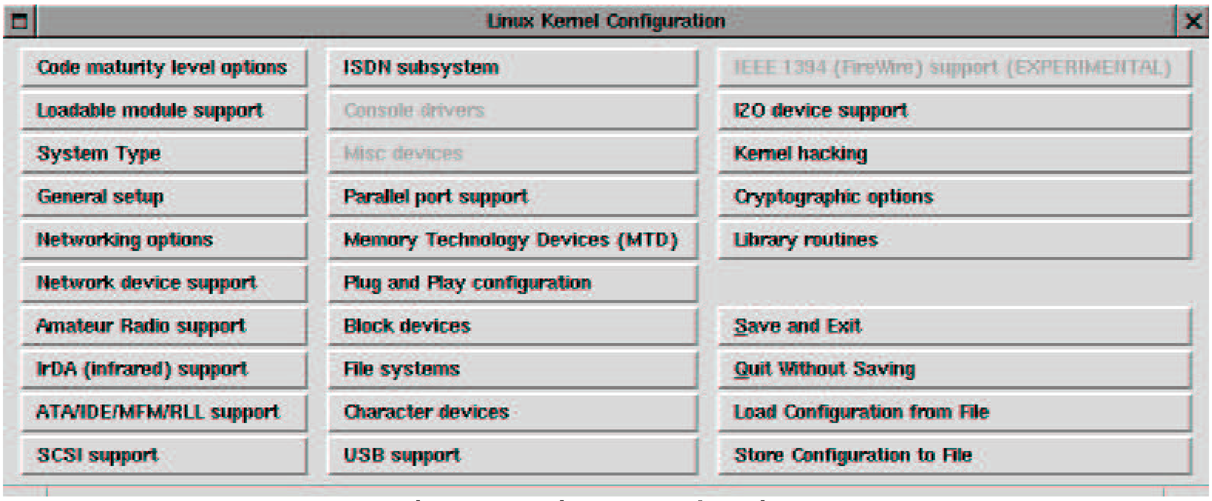

Figure 2-1: Linux Kernel settings

After you have pressed "Save and Exit", the Kernel configuration (Figure 2-1) resp. the Application configuration (Figure 2-2) dialog will open.

Further information about the distribution itself can be found in the file README and in the Documentation directory.

[1] Sometimes a number of questions will appear after you 'Save and Exit'. Do not be concerned, it just means that some new config options have been added to the source tree that do not have defaults for the configuration you have chosen. If this happens the safest option is to answer 'N' to each question as they appear.

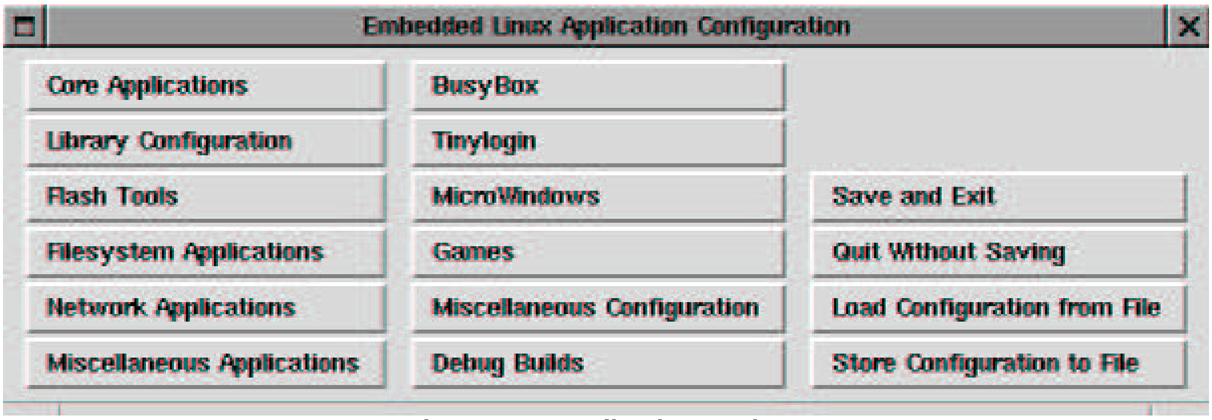

Figure 2-2: Application settings

2.1. Compiler

The cross compiler toolchain contains a GNU-C compiler version 2.95.3. It will generate code for the target platform arm-elf. The toolchain package comes as a shell script with an attached tar archive. To install it, log in as root and type

```
sh arm-elf-tools-20030314.sh
```

This will install the cross compiler toolchain in its standard path `/usr/local/`. Make sure `/usr/local/bin` is in your shell's search path. To deinstall the toolchain, just delete the installed files:

```
rm -rf "/usr/local/arm-elf"
rm -rf "/usr/local/lib/gcc-lib/arm-elf"
rm -f /usr/local/bin/arm-elf-*
```

2.2. Library

uClibc (aka µClibc/pronounced yew-see-lib-see) is a C library for developing embedded Linux systems. It is much smaller than the GNU C Library, but nearly all applications supported by glibc also work perfectly with uClibc. Porting applications from glibc to uClibc typically involves just recompiling the source code.

3. The Linux Kernel

The *uClinux-dist* package comes with three different kernels, versions 2.0.x, 2.4.x, and 2.6.x (http://www.uclinux.org/). We are using the kernel version 2.4. The kernel source is based on the standard Linux kernel 2.4.24. It contains the necessary µClinux patches, i.e. it is already modified to run on processors without *Memory Management Unit* (MMU).

3.1. Hardware Specific Modifications

This section will describe the hardware specific modifications of the kernel sources made for the System-on-Chip Lite+ board.

Hardware specific header files for the System-on-Chip Lite+ system are located in the subdirectory `include/ asm-armnommu/arch-socltplus`. Hardware specific new source files (apart from device drivers) can be found in `arch/armnommu/mach-socltplus/`.

3.1.1. Startup Code

Many CPU's start execution at a fixed address (for example ARM, x86). Others read a fixed location in the address space and use that value as the start address (m68k, ColdFire). System-on-Chip Lite+, like most ARM CPUs, starts executing at offset 0.

To be able to bootstrap, the kernel code must be located in the Flash/ROM memory connected to chip select signal nXCSS1, and the EA pins must be set to `EA(1:0) = 01`. The µClinux startup code has to do all hardware setup, this usually includes setting up chip selects and RAM setup. It then loads and starts execution of the µClinux kernel properly. The only difficulty with this scheme is arranging the kernel code to be at the correct offset in the Flash memory so that the CPU will start executing it on reset. Special care has to be taken if the kernel code is moved before booting, e.g. copied from Flash to RAM. Any code executed before this relocation has to be position independent.

Linux' entry point is usually defined in a file named *head*, found in a directory path of pattern `arch/ <cpu family>/kernel/head_<cpu type>.S`. The startup code for System-on-Chip Lite+ is located in `arch/armnommu/kernel/head_socltplus.S`.

At bootup, it first checks the Remap Controller. If the Remap bit is already set, it is assumed that the kernel was downloaded into memory by a debug monitor and the memory has already been initialized. If the Remap bit is not set (it is cleared by a reset), the memory controller will be set according to the settings in `soc_sysconf.h` and the SDRAM will be initialized. Settings regarding SDRAM are defined in `soc_sdram-config.h`. Next, the kernel code will be copied into RAM, and execution continues there.

Also, the UART will be configured for debug output. Eventually, a jump to *start_kernel()* starts the Linux kernel itself.

3.1.2. Interrupts and Other Exceptions

Other hardware specific adaptations had to be done for the interrupt and exception handling. These routines can be found in a directory path made from the pattern

```
arch/<cpu family>/kernel/entry_<cpu type>.s
```

The file concerning the System-on-Chip Lite+ is

`entry_armv.S`

One important modification is a new macro definition for `get_irqnr_and_base`, a code sequence that reads the appropriate interrupt controller register and identifies the interrupt source.

The first section of the internal RAM mapped to address 0 is used for the exception vectors. After the vectors and stubs are copied there in the subroutine `trap_init`, this area, WPA0, will be write protected using System-on-Chip Lite+'s Write Protection controller.

More hardware specific functions for enabling, disabling and acknowledging IRQs are defined in file `arch/armnommu/mach-socltplus/irq.c`.

3.1.3. Low Level Debug Output

The code in `head-arm-socltplus.S` will output some debug messages on the serial interface during startup, if `CONFIG_SOC_BOOTLOADER_DEBUG` is defined in this file. It also uses the 7 segment LED displays to show codes like "A1", "C3" etc. that could help identify the position if the startup gets stuck. More diagnostic serial output is activated by defining `CONFIG_SOC_BOOTLOADER_VERBOSE`.
If both constants are undefined, the startup is (almost) quiet.
Since the startup code makes use of the low level output routines in `arch/armnommu/kernel/debug-armv.S` be sure to select "Kernel low-level debugging functions" (`CONFIG_DEBUG_LL`) in the kernel configuration dialog. The default baud rate for these low level functions and for the console output is defined in `soc_sysconf.h` (constant `DEFAULT_BAUD_RATE`).

3.2. New or Modified Drivers

3.2.1. Serial Interface

The serial driver `serial_socltplus.c` is based on the standard driver `serial.c`. As the System-on-Chip Lite+ UART only supports 8 data bits, 1 stop bit and no parity, differing settings in `ioctl()` and related functions are ignored or return error codes. The driver supports baud rates from 600bd to 115200bd.

The System-on-Chip Lite+ serial driver should be activated in the "Character Devices" kernel configuration dialog. Optionally, the serial port can be used as the system console (the system console is the device which receives all kernel messages and warnings and which allows logins in single user mode). This could be useful if some terminal or printer is connected to that serial port.

You can define a kernel command line option to select which device to use for console output. The format of this option is:

```
console=device,options

device:     tty0  for the foreground virtual console
            ttySx for a serial port

options:    depend on the driver. For the serial port this
            defines the baudrate/parity/bits of the port, in
            the format BBBBPN, where BBBB is the speed, P is
            parity (n/o/e), and N is bits.
```

3.2.2. Ethernet

As the System-on-Chip Lite+ uses a Cadence MACB Ethernet controller, the MACB driver supplied by Cadence is integrated in the kernel. The sources have been modified for the System-on-Chip Lite+ hardware, and can be found in the subdirectory `drivers/net/cadence_macb/`. The controller's base address is adapted and the driver now can allocate buffer memory dynamically. If the constant `MACB_SHARED_MEM_BASE` is defined in file `macb_linux_ext.h`, a special fixed buffer at this address will be used. If the constant is undefined, the buffer will be allocated at startup using `kmalloc()`.

The MAC address of the Cadence MACB Ethernet controller is generated from a base address, `ETH_MACB_BASE_ADDR`, defined in soc_sysconf.h, plus the lower 24 bits of the boards serial number (see chapter 4.6.2).

3.2.3. MTD Flash Mapping

The MTD drivers clearly provide the most powerful support for Flash, so they normally are to be preferred over blkmem or FTL drivers. They also allow you to run real read/write filesystems specifically designed for Flash memory, such as JFFS and JFFS2 [11].

The Linux MTD drivers support a huge variety of Flash devices, and offer powerful mechanisms for defining partitions and mappings. For anything other than trivial setups you create a map driver that defines your exact Flash layout. It can span multiple Flash devices, with interleaving, and even different Flash device types in the one system. Most importantly, you can partition a physical Flash device into several logical devices.

These mapping drivers are located in the subdirectory `drivers/mtd/maps`. Mappings for the System-on-Chip Lite+ board can be found in the files `socltplus-flash.c` and `socltplus-cs0flash.c`, the latter contains an example for the currently unused flash memory at nXCSS0. Additions in the `Makefile` and `Config.in` provide the respective options in the configure dialog.

The most important part of a mapping file is the definition of the partition structure:

```
static struct mtd_partition soc_cs1partitions[3] = {
      {
            name:       "SOClite+ Kernel",
            offset:     0x00000000,
            size:       0x00180000  /* 1.5 MB should be more than enough */
      },
      {
            name:       "SOClite+ Filesystem",
            offset:     0x00180000,
            size:       0x00660000  /* almost 6.4 MB for files */
      },
      {
            name:       "SOClite+ Config",
            offset:     0x007e0000,
            size:       0x00020000  /* some room for configurations, */
                      /* "Top" flash has 8 sectors of 8 KB each here */
      }
};
```

This structure creates 3 Flash partitions. The memory chips on the System-on-Chip Lite+ board, AMD 29DL323GT, have erase sectors of 64 KB size. The bank at chip select signal nXCSS1 uses two chips in parallel, making the logical erase sector size 128 KB. Flash partitions must begin on sector boundaries, if you want them to be writeable.

The first partition in this example, `/dev/mtd0`, holds the kernel in uncompressed state. The second partition, `/dev/mtd1` resp. `/dev/mtdblock1`, has about 6.375 MB room for a root file system like JFFS2 or ROMFS.

The third partition, `/dev/mtd2`, comprises the area of smaller erase sectors (see data sheet [2]), and can be used for configuration data.

3.2.4. Watchdog

Usually a userspace daemon will notify the kernel watchdog driver via the `/dev/watchdog` special device file that userspace is still alive, at regular intervals. When such a notification occurs, the driver will usually tell the hardware watchdog that everything is in order, and that the watchdog should wait for yet another little while to reset the system. If userspace fails (RAM error, kernel bug, whatever), the notifications cease to occur, and the hardware watchdog will reset the system (causing a reboot) after the timeout occurs.

In the directory `linux-2.4.x/drivers/char/` you find a watchdog driver for the System-on-Chip Lite+ system, `wdt_socltplus.c`. This driver makes use of the System-on-Chip Lite+'s watchdog timer hardware and provides support for the device `/dev/watchdog`.

The timeout period of the watchdog timer hardware in the System-on-Chip Lite+ system is a function of HCLK (50 MHz) and the WD reload value in the Watchdog Timer Control register. As WDRV is only 12 bits, the maximum interval is approximately 1.3 seconds. Linux watchdog daemons often expect trigger intervals of about 30 or 60 seconds. So the idea behind this driver is to ping the watchdog hardware regularly from a timer function, until the software interval runs out. Soft- and hardware trigger intervals are configurable in the source code.

3.2.5. LED

The same subdirectory holds the LED driver led_socltplus.c. It can output one or two characters, optionally followed by dots, on the System-on-Chip Lite+ board's 7-segment LED displays. It supports numbers 0-9, letters A-Z (unfortunately not all of them can be assigned in a unique way to legible 7-segment patterns), brackets, "-" and "=".

Also see chapter 6.

4. Userland Application Programs

4.1. Init

`Init` is the parent of all processes. Its primary role is to create processes from a script stored in the file `/etc/inittab`. This file usually has entries which cause init to spawn gettys on each line that users can log in. It also controls autonomous processes like server daemons required by any particular system.

4.2. BusyBox

`BusyBox` combines tiny versions of many common UNIX utilities into a single small executable. It provides minimalist replacements for most of the utilities you usually find in GNU coreutils, util-linux, etc. The utilities in BusyBox generally have fewer options than their full-featured GNU cousins; however, the options that are included provide the expected functionality and behave very much like their GNU counterparts [9].

BusyBox has been written with size-optimization and limited resources in mind. It is also extremely modular so you can easily include or exclude commands (or features) at compile time. This makes it easy to customize your embedded systems. BusyBox provides a fairly complete POSIX environment for any small or embedded system.

BusyBox is extremely configurable. This allows you to include only the components you need, thereby reducing binary size. Run `make xconfig` or `make menuconfig` to select the functionality that you wish to enable.

4.3. Boa

Boa is a single-tasking HTTP server. That means that unlike traditional web servers, it does not fork for each incoming connection, nor does it fork many copies of itself to handle multiple connections. It internally multiplexes all of the ongoing HTTP connections, and forks only for CGI programs (which must be separate processes), automatic directory generation, and automatic file gunzipping.

The primary design goals of Boa are speed and security. Security, in the sense of "can't be subverted by a malicious user," not "fine grained access control and encrypted communications". Boa is not intended as a feature-packed server; if you want one of those, check out WN from John Franks.

4.3.1. Files Used by Boa

- `boa.conf`
  This file is the sole configuration file for Boa. The directives in this file are defined in the DIRECTIVES section.
- `mime.types`
  The MimeTypes <filename> defines what Content-Type Boa will send in an HTTP/1.0 or better transaction.

4.3.2. boa.conf Directives

The Boa configuration file is parsed with a lex/yacc or flex/bison generated parser. If it reports an error, the line number will be provided; it should be easy to spot. The syntax of each of these rules is very simple, and they can occur in any order.

Note: the "ServerRoot" is not in this configuration file. It can be compiled into the server (see `defines.h`) or specified on the command line with the -c option.

The following directives are contained in the boa.conf file, and most, but not all, are required.

`Port <integer>`
This is the port that Boa runs on. The default port for http servers is 80. If it is less than 1024, the server must be started as root.

`Listen <IP>`
Listen: the Internet address to bind(2) to. If you leave it out, it takes the behaviour before 0.93.17.2, which is to bind to all addresses (INADDR_ANY). You only get one "Listen" directive, if you want service on multiple IP addresses, you have three choices:

1. Run boa without a "Listen" directive
   (a) All addresses are treated the same; makes sense if the addresses are localhost, ppp, and eth0.
   (b) Use the VirtualHost directive below to point requests to different files. Should be good for a very large number of addresses (web hosting clients).
2. Run one copy of boa per IP address, each has its own configuration with a "Listen" directive. No big deal up to a few tens of addresses. Nice separation between clients. The name you provide gets run through inet_aton(3), so you have to use dotted quad notation. This configuration is too important to trust some DNS.

`User <user name or UID>`
The name or UID the server should run as. For Boa to attempt this, the server must be started as root.

`Group <group name or GID>`
The group name or GID the server should run as. For Boa to attempt this, the server must be started as root.

`ServerAdmin <email address>`
The email address which server problems should be sent to. Note: this is not currently used.

`ErrorLog <filename>`
The location of the error log file. If this does not start with /, it is considered relative to the server root. Set to `/dev/null` if you don't want errors logged.

`AccessLog <filename>`
The location of the access log file. If this does not start with /, it is considered relative to the server root. Comment out or set to `/dev/null` (less effective) to disable access logging.

`VerboseCGILogs`
This is a logical switch and does not take any parameters. Comment out to disable. All it does is switch on or off logging of when CGIs are launched and when the children return.

`CgiLog <filename>`
The location of the CGI error log file. If specified, this is the file that the stderr of CGIs is tied to. Otherwise, writes to stderr meet the bit bucket.

`ServerName <server_name>`
The name of this server that should be sent back to clients if different than that returned by gethostname.

`VirtualHost`
This is a logical switch and does not take any parameters. Comment out to disable. Given DocumentRoot /var/www, requests on interface `A' or IP `IP-A' become `/var/www/IP-A`. Example: `http://localhost/` becomes `/var/www/127.0.0.1`

`DocumentRoot <directory>`
The root directory of the HTML documents. If this does not start with /, it is considered relative to the server root.

`UserDir <directory>`
The name of the directory which is appended onto a user's home directory if a user request is received.

`DirectoryIndex <filename>`
Name of the file to use as a pre-written HTML directory index. Please make and use these files. On the fly creation of directory indexes can be slow.

`DirectoryMaker <full pathname to program>`
Name of the program used to generate on-the-fly directory listings. The program must take one or two command-line arguments, the first being the directory to index (absolute), and the second, which is optional, should be the "title" of the document be. Comment out if you don't want on the fly directory listings. If this does not start with /, it is considered relative to the server root.

`DirectoryCache <directory>`
DirectoryCache: If DirectoryIndex doesn't exist, and DirectoryMaker has been commented out, the on-the-fly indexing of Boa can be used to generate indexes of directories. Be warned that the output is extremely minimal and can cause delays when slow disks are used. Note: The DirectoryCache must be writable by the same user/group that Boa runs as.

`KeepAliveMax <integer>`
Number of KeepAlive requests to allow per connection. Comment out, or set to 0 to disable keepalive processing.

`KeepAliveTimeout <integer>`
Number of seconds to wait before keepalive connections time out.

`MimeTypes <file>`
The location of the mime.types file. If this does not start with /, it is considered relative to the server root. Comment out to avoid loading mime.types (better use AddType!)

`DefaultType <mime type>`
MIME type used if the file extension is unknown, or there is no file extension.

`AddType <mime type> <extension> extension...`
Associates a MIME type with an extension or extensions.

`Redirect, Alias`, and `ScriptAlias  <path1>  <path2>`
Redirect, Alias, and ScriptAlias all have the same semantics - they match the beginning of a request and take appropriate action. Use Redirect for other servers, Alias for the same server, and ScriptAlias to enable directories for script execution.

4.4. MTD-Utilities

This directory holds a variety of utilities for creating filesystems, testing flash device integrity and other stuff dealing with Memory Technology Devices.

In particular we find `mkfs.jffs2` here, a tool to create JFFS2 (Journaling Flash File System) images. As the file system images are built on the host, these tools are built for the host machine too (subdirectory `build`).

4.5. SMTP-Client

The program `mail` is a minimal SMTP client that takes an email message body and passes it on to a SMTP server (default is the MTA on the local host). Since it is completely self-supporting, it is especially suitable for use in restricted environments [10].

```
Message Header Options:
-s,    --subject=STR        subject line of message
-f,    --from=ADDR          address of the sender
-r,    --reply-to=ADDR      address of the sender for replies
-e,    --errors-to=ADDR     address to send delivery errors to
-c,    --carbon-copy=ADDR   address to send copy of message to

Processing Options:
-S,    --smtp-host=HOST     host where MTA can be contacted via SMTP
-P,    --smtp-port=NUM      port where MTA can be contacted via SMTP
-M,    --mime-encode        use MIME-style translation to quoted-printable
-L,    --use-syslog         log errors to syslog facility instead of stderr

Giving Feedback:
-v,    --verbose            enable verbose logging messages
-V,    --version            display version string
-h,    --help               display this page
```

4.6. MTDW

This directory holds two small programs that make writing to a `Memory Technology Device` easier.

4.6.1. mtdw

The MTD writer program `mtdw` can write an arbitrary file to a logical MT device, like a flash memory partition.

```
mtdw [-e] <file> <mt device>
```

Normally, `mtdw` will erase only as many blocks of the device as needed for `file`. With the option `-e`, `mtdw` erases the whole (logical) device. This is necessary when an JFFS2 file system image is to be written:

```
mtdw -e jffs2.img /dev/mtd1
```

The flash memory partitioning is defined in an mtd mapping file (3.2.3). For the System-on-Chip Lite+ board, this is `drivers/mtd/maps/socltplus-flash.c`. The kernel itself resides in `/dev/mtd0`, the JFFS2 file system in `/dev/mtd1`, and optional configuration data in `/dev/mtd2` (mtd2 is the area of smaller erase sectors, see data sheet [2]).

4.6.2. ksetup

The program ksetup can write some configuration data - a serial number and a kernel command line - as a `tagged list` into an MTD partition. The kernel can parse this list at startup and set up the respective system parameters.

To tell the kernel where to look for this list, the constant `SOC_BOOTPARAMS` has to bet set to the appropriate value (here: 0x047e0000) before compilation (see file `include/asm-armnommu/arch-socltplus/soc_sysconf.h`).

The MAC address of the Cadence MACB Ethernet controller will be generated from ETH_MACB_BASE_ADDR, defined in soc_sysconf.h, plus the lower 24 bits of the serial number (if available).

```
ksetup <rom-device> [<serial high> <serial low> <root dev. id> <command line>]
```

```
<rom-device>:    MTD device the parameter list will be written to, e.g.
                 /dev/mtd2
<serial high>:   higher part (32 bit) of the 64-bit serial number
<serial low>:    lower part of the serial number
<root dev. id>:  ID of the root device, e.g. 0x0300 (ram0), 0x1f01
                 (mtdblock1)
<command line>:  The kernel command line, e.g. "console=/dev/ttyS0,9600"
```

Example of a kernel command line (it should actually be one line, linebreak inserted for readability only):

```
console=ttyS0,38400 root=/dev/mtdblock1
 ip=192.168.30.21::192.168.30.254:255.255.255.0:socltplus::off
```

Keywords:

```
console=port, baudrate
root=rootpartition
ip=own IP :server IP :gateway :netmask :hostname:net device:dynamic configuration
```

Arguments for ip may be empty, trailing empty arguments can be left off. Another example, to mount a root file system via NFS:

```
console=ttyS0,38400 root=/dev/nfs nfsroot=/LinuxSOC/rootfs
 ip=192.168.30.21:192.168.30.254:192.168.30.254:255.255.255.0:socltplus
```

Note that this network configuration might be overwritten later by the startup script rc, where the default configuration defined in `vendors/NEC/SOClite+/Makefile` is set.

[MEMO]

5. Debugger

5.1. Insight

`Insight` is a version of GDB, the GNU Debugger, that uses Tcl/Tk to implement a graphical user interface. It is a fully integrated GUI, not a separate front-end program. The interface consists of several separate windows, which use standard elements like buttons, scrollbars, entry boxes and such to create a fairly easy to use interface. Each window has a distinct content and purpose, and can be enabled or disabled individually. The windows contain things like the current source file, a disassembly of the current function, text commands (for things that aren't accessible via a button), and so forth.

5.1.1. Building Insight

Building Insight is very straightforward (it is configured by default when you checkout or download Insight). Right now, Insight must be built using the versions of Tcl, Tk, Itcl, and Tix that come with the sources.

```
tar xzvf insight-5.2.1.tar.gz
cd insight-5.2.1
./configure --target=arm-elf-uclinux
make
```

The new built program could be executed directly from its directory`, insight-5.2.1/gdb/gdb`. Of course you can install insight in your system; you will need root permissions to do that:

```
cd insight-5.2.1
make install
```

5.1.2. Using Insight

Just run it like you would a normal GDB (in fact, it's actually called 'gdb'). If everything goes well, you should have several windows pop up. To get going, hit the Run button, and go exploring.

If you want to use GDB in command line mode, just use the -nw option. Or, you can undefine the DISPLAY environment variable.

Insight comes with all your standard debugger windows, including:

- Console Window
- Source Window
- Register Window
- Memory Window
- Locals Window
- Watch Window
- Stack Window
- Thread/Process Window
- Function Browser Window
- Debug Window (for developers)

Insight also has an extensive (if outdated) online help system which describes all the windows and explains how to use them. Users are urged to browse this help system for information on using Insight.

If a script file named `.gdbinit` is present in the current directory, it will be executed when GBD/Insight is started. The `linux-2.4.x` directory contains an init file for the System-on-Chip Lite+ system that sets up the memory controller and initializes the DRAM.

5.2. *gdbserver*

GDBSERVER is a program that allows you to run GDB on a different machine than the one which is running the program being debugged.

5.2.1. Usage on Target Side

First, you need to have a copy of the program you want to debug put onto the target system. The program can be stripped to save space if needed, as gdbserver doesn't care about symbols. All symbol handling is taken care of by the GDB running on the host system.

To use the server, you log on to the target system, and run the 'gdbserver' program. You must tell it

(a) how to communicate with GDB,
(b) the name of your program, and
(c) its arguments.

The general syntax is:

```
target> gdbserver COMM PROGRAM [ARGS...]
```

For example, using a serial port, you might say:

```
target> gdbserver /dev/com1 emacs foo.txt
```

This tells gdbserver to debug emacs with an argument of foo.txt, and to communicate with GDB via /dev/com1. Gdbserver now waits patiently for the host GDB to communicate with it.
To use a TCP connection, you could say:

```
target> gdbserver host:2345 emacs foo.txt
```

This says pretty much the same thing as the last example, except that we are going to communicate with the host GDB via TCP. The `host:2345` argument means that we are expecting to see a TCP connection from 'host' to local TCP port 2345. (Currently, the 'host' part is ignored.) You can choose any number you want for the port number as long as it does not conflict with any existing TCP ports on the target system. This same port number must be used in the host GDB's 'target remote' command, which will be described shortly. Note that if you chose a port number, that conflicts with another service, gdbserver will print an error message and exit.

5.2.2. Usage on Host Side

You need an unstripped copy of the target program on your host system, since GDB needs to examine its symbol tables and such. Start up GDB as you normally would, with the target program as the first argument. (You may need to use the `--baud` option if the serial line is running at anything except 9600 baud.) Ie: `gdb TARGET-PROG`, or `gdb --baud BAUD TARGET-PROG`. After that, the only new command you need to know about is 'target remote'. Its argument is either a device name (usually a serial device, like `/dev/ttyb`), or a HOST:PORT descriptor. For example:

```
(gdb) target remote /dev/ttyb
```

communicates with the server via serial line `/dev/ttyb`, and:

```
(gdb) target remote the-target:2345
```

communicates via a TCP connection to port 2345 on host 'the-target', where you previously started up gdbserver with the same port number. Note that for TCP connections, you must start up gdbserver prior to using the 'target remote' command, otherwise you may get an error that looks something like 'Connection refused'.

When the connection is established, you should start the execution of your program by selecting "continue", not "run". Sometimes the source window is blanked after the connection. The source text will reappear at the first breakpoint.

5.2.3. Options

You have to supply the name of the program to debug and the tty to communicate on; the remote GDB will do everything else. Any remaining arguments will be passed to the program verbatim.

[MEMO]

6. Adding Kernel Drivers

Development of Linux kernel drivers can be an extensive task. If you seriously want to delve into it, we recommend consulting a good book on the topic, like [7].

But for the really adventurous, here's the short version. We will use the simple LED driver mentioned in chapter 3.2.5 as an example. As µClinux for the ARM-NoMMU architecture does not (yet) support loadable kernel modules, we will integrate the driver in the kernel source tree. All paths given in this chapter are relative to the kernel source directory, `uClinux-dist/linux-2.4.x/`.

6.1. Write the Driver

Since our example device is a character-related device, we will add the driver to the directory `drivers/char/led_socltplus.c`.

Have a look at the source code: The functions `led7seg_init()` and `led7seg_exit()` are declared as `module_init` resp. `module_exit`. This makes sure they are called automatically at startup resp. shutdown, even if the driver is not a separate module.

In `led7seg_init()` our driver is registered as "miscellaneous" device, which seems appropriate for a driver like this. Two important structures have to be declared:

```
static struct file_operations soc_led_fops = {
      owner:      THIS_MODULE,
      read:       soc_led_read,
      write:      soc_led_write,
      ioctl:      soc_led_ioctl,
      open:       soc_led_open,
      release:    soc_led_release,
};

static struct miscdevice soc_led_miscdev= {
      minor:      LED_MINOR,
      name:       "led7seg",
      fops:       &soc_led_fops,
};
```

The miscdevice structure is passed to the registering function and informs it about name, file operation functions and minor number. The minor number in this case is 151, the major number for misc devices is always 10.

In the file operations structure you have to define your functions for open and release (close). The others are optional, but obviously you need at least one of them to make the driver do anything useful.

The primary function here is `soc_led_write`, which reads up to four characters from user space and sets the LED output registers accordingly. It is recommended to use the `get_user` or `copy_from_user` functions to read data from the user process, even under µClinux.

6.2. Add a Configuration Option

See `drivers/char/Config.in`, line 225. This driver works only on the NEC System-on-Chip Lite+ board, so we add the option in the respective *`if`*-section:

```
if [ "$CONFIG_MACH_SOCLTPLUS" = "y" ]; then
   bool 'SOClite+ serial port support' CONFIG_SERIAL_SOCLTPLUS
   if [ "$CONFIG_SERIAL_SOCLTPLUS" = "y" ]; then
      bool 'Console on SOClite+ serial port' CONFIG_SERIAL_SOCLTPLUS_CONSOLE
   fi
   bool 'SOClite+ 7-segment LED support' CONFIG_SOCLTPLUS_LED
fi
```

6.3. Add a Makefile Entry

Now we have to make sure the driver is compiled and linked to the kernel if the option `CONFIG_SOCLTPLUS_LED` is set to "y". We add a line to drivers/char/Makefile (line 238):

```
obj-$(CONFIG_SOCLTPLUS_LED) += led_socltplus.o
```

By this, `led_socltplus.o` is added to the `obj-y` list if selected. The Makefile rules take care of the rest.

6.4. Add a Device Node

A user application typically makes use of a kernel driver by opening a device file and writing to or reading from it. These device nodes are files of a special type. By convention, they are kept in the `/dev` directory. From a command line, device files can be created using the command `mknod <filename> <type> <major> <minor>`, where filename is the device file to be created, type is 'c' for a character device or 'b' for a block device, and major and minor are the major and minor device numbers.

To create the device node in the target file system each time it is built, you need to add it to the device list in the vendor Makefile, located in `uClinux-dist/vendors/NEC/SOClite+/` (see line 65).

```
watchdog,c,10,130 led,c,10,151 \
```

Besides the watchdog device, this line creates a character device node named "led" with the major number 10 and the minor number 151.

Now the new driver can be accessed from user level. A very simple display test would be redirecting the keyboard input to the LED device. On the console, type `cat >/dev/led` and hit *return*, then one or two alphanumeric characters, again followed by *return*. Stop the *cat* process with *Control-c*.

7. Adding User Applications

This chapter gives simple instructions for adding a user-written application to the µClinux configuration system.

7.1. General Approach

Entries must be added to three files, and an appropriate Makefile must exist in the user application source directory, which must be put in `user/` (all directory names here are given relative to the µClinux top directory `uClinux-dist/`).

Files to edit:

- `user/Makefile`

  Add a line to the file like

  ```
  dir_$(CONFIG_USER_FOO_FOO)  += foo
  ```

  This adds the directory 'foo' to the list of directories to be built. I added mine in alphabetical order. The order doesn't seem to matter.

- `config/Configure.help`

  This file contains the text which is presented on request during the config. Add a block like

  ```
  CONFIG_USER_FOO_FOO
    This program does fooey things to your bars.
  ```

  The text must be indented two spaces, and there must be no empty lines. Lines should be < 70 chars long.

- `config/config.in`

  Add a line in the appropriate menu section (i.e. in the program group you want your app to show up in during 'make config'; I used 'misc'), like

  ```
  bool 'foo'      CONFIG_USER_FOO_FOO
  ```

  The repetition of FOO allows for directories which contain multiple executables. Thus, if the user directory 'foo' contained code to make 'foo' and 'bar', each gets its own config line if an additional entry is made like

  ```
  bool 'bar'      CONFIG_USER_FOO_BAR
  ```

Next, there needs to be a proper `user/foo/Makefile`. The Makefile should follow the following template:

```
EXEC = foo
OBJS = foo.o

all: $(EXEC)

$(EXEC): $(OBJS)
	$(CC) $(LDFLAGS) -o $@ $(OBJS) $(LDLIBS)

romfs:
	$(ROMFSINST)	/bin/$(EXEC)

clean:
	-rm -f $(EXEC) *.elf *.gdb *.o
```

If more than one executable is built in the foo directory, as above, then the Makefile should look like

```
EXECS = foo bar
OBJS = foo.o bar.o

all: $(EXECS)

$(EXECS): $(OBJS)
     $(CC) $(LDFLAGS) -o $@ $@.o  (LDLIBS)

romfs:
     $(ROMFSINST) -e CONFIG_USER_FOO_FOO      /bin/foo
     $(ROMFSINST) -e CONFIG_USER_FOO_BAR      /bin/bar
```

More complex makefiles are of course possible. The reader is encouraged to browse the user tree for examples.

When all this is set up, doing the standard `make xconfig; make dep; make` should build the application and install it in romfs and hence in the target system image.bin.

7.2. LED Sample Application

As an example we will use a little program that scrolls a text through the LED display.

```
#include <stdio.h>
#include <unistd.h>
#include <string.h>
#include <fcntl.h>

int main (int argc, char *argv[])
{
      int i, l, f;

      if (argc != 2) {
            puts("Usage: ledscroll <text>\n");
            return 1;
      }
      f = open("/dev/led", O_WRONLY);
      l = strlen( argv[1] );
      if (l <= 2) {
            write(f, argv[1], l);
      } else {
            for ( i = 0; i <(l-1); i++ ) {
                  write(f, &(argv[1][i]), 2);
                  sleep(1);
            }
      }
      close(f);
      return 0;
}
```

To test compile this program before integrating it into the µClinux-dist build system, you have to call the compiler with a lot of options to make it find the correct startup code, include files and libraries (all on one line, of course):

```
arm-elf-gcc -mcpu=arm7tdmi -I/LinuxSOC/uClinux-dist/lib/uClibc/include
 -fno-builtin -nostartfiles -Wl,-elf2flt
 /LinuxSOC/uClinux-dist/lib/uClibc/lib/crt0.o
 /LinuxSOC/uClinux-dist/lib/uClibc/lib/crti.o
 /LinuxSOC/uClinux-dist/lib/uClibc/lib/crtn.o  -Wl,-move-rodata
 -L/LinuxSOC/uClinux-dist/lib/uClibc/lib  -o  ledscroll  ledscroll.c
```

In the build system, these options are generated automatically. Note that `/LinuxSOC/uClinux-dist` is the path to µClinux-dist on this test machine. If yours is different, adapt the path parameters.

Once the program is tested, we can integrate it into the userland tree, as explained in the previous section. The source code is installed in the new directory `users/ledscroll`. A short Makefile is added:

```
EXEC = ledscroll
OBJS = ledscroll.o

all: $(EXEC)

$(EXEC): $(OBJS)
	$(CC) $(LDFLAGS) -o $@ $(OBJS) $(LDLIBS)

romfs:
	$(ROMFSINST) /bin/$(EXEC)

clean:
	-rm -f $(EXEC) *.elf *.gdb *.o
```

Also, we add entries in users/Makefile (Line 145),

```
dir_$(CONFIG_USER_LEDSCR_LEDSCR)     += ledscroll
```

and in config/config.in (Line 519):

```
bool 'ledscroll'                    CONFIG_USER_LEDSCR_LEDSCR
```

To greet the user with a message at startup, the startup script has to be modified. This startup script, rc, is generated dynamically by vendors/NEC/SOClite+/make-rc. The lines

```
if [ "$CONFIG_USER_LEDSCR_LEDSCR" ]; then
  invis_init_cmd 'ledscroll "dont panic"'
fi
```

will add a command to the rc script, if the *`ledscroll`* application is selected in the configuration.

[MEMO]

8. Notes

The µClinux system is a port of regular Linux to microprocessors that lack a memory management unit (MMU). The implications of this on µClinux are primarily that there is no memory protection (processes can write anywhere in memory), and no virtual memory (swapping etc). For most user applications the only implication is that the *fork()* system call is unavailable, and *vfork()* must be used instead.

8.1. Memory Access Without MMU

The MMU normally provides a level of protection for applications that run on the platform. Working without the MMU means that all of the program memory is literally mapped against the physical memory. This is called a flat memory architecture. An invalid memory pointer in a user application may trigger an address error which can completely freeze or corrupt an MMU-less system. The code implemented in MMU-less platforms has to be working properly, which of course means thorough testing.

Dynamic memory allocation in a flat memory model can also cause fragmentation which can starve the system. In an ideal case the physical memory is used as continuous memory areas and the allocation should fail only when the number of free page frames is too small. Otherwise, there might not be a continuous piece of memory available, although the number of free memory pages in total would be large enough.

To ease this problem, in the µClinux kernel an optional memory allocating strategy named *Non power-of-2 kernel allocator* can be activated. It offers allocations in more flexible block sizes, at the price of a slight performance loss.

8.2. Creating New Processes

In standard Linux, the fork() system call is used to duplicate the current process by creating a new entry in the process table. This can be handy if the program handles more than one function at the time. The created child process is almost identical to the parent executing the same code but with its own data space, environment and descriptors. The fork command is implemented by using *copy-on-write* pages. When the child process tries to write to the page frame, a private copy of the page is created for this process. The new physical page is mapped into the original logical address space. Without MMU the system cannot completely and securely clone a process, nor does it have access to copy-on-write pages.

The µClinux implements BSDs *vfork()* in order to offer similar functionality. The process created by this system call shares all their memory space including the stack. To prevent the parent from overriding the data needed by the child process the parent is suspended until the child exits.

8.3. File Systems

There are a number of choices for root filesystem in µClinux.

Traditionally the ROMfs type has been the most commonly used. It is a simple, compact, read-only filesystem. It stores all data of a file sequentially, so it allows for application programs to be executed in place (XIP) in the filesystem on µClinux targets that support this. This can make for a considerable reduction in memory footprint for a running system.

Cramfs is a new filesystem for 2.4 series Linux kernels. It is designed to be a compact read only filesystem. Its primary advantage is that it stores all files compressed and decompresses them on the fly. Because it store files compressed, you cannot run applications in place (no XIP). It is quite space efficient in terms of Flash usage, but more RAM will be required since all application code needs to be copied into RAM for execution.

Some systems will need a read/write root filesystem. By using the Linux MTD drivers it is possible to run a journaled Flash filesystem like JFFS or JFFS2 on top of Flash memory. Journaled filesystems are safe from sudden power loss (that is an unclean shutdown condition), and don't require a filesystem check on the next boot up. Since the JFFS and JFFS2 filesystems are specifically designed for use with Flash memory, they also provide a feature called “wear levelling”. This is where the filesystem code lays out data and updates it in such a way that all parts of the Flash are erased a similar number of times. This can dramatically increase the useful lifetime of Flash memory devices. JFFS2 has the distinct advantage of storing files compressed, so uses much less Flash space. It should be used in preference to the older JFFS. Something else to be mindful of when using a journaled filesystem is that some small amount of Flash will be wasted for the journal overhead and garbage collection system. This wasted space is typically of the order of two Flash segments in size.

9. Tips and Tricks

9.1. *Mounting an nfs Network Drive*

For stable and reliable operation network drive usage, the *mount* should be used with mount options. Without options, the network connection might get stuck.

Following command line in the uCLinux terminal window resulted in a stable operation:

mount -t nfs -o nolock,rsize=1024,wsize=1024 <network dir> /mnt

Sample for <network dir>: 172.29.29.156:/home/kaiserr/temp

9.2. *Fast Update of µcLinux Kernel and JFFS2 File System*

Pre-Requisite is an established network connection to a host with the binary images of the kernel and JFFS2 file system (See 9.1). In that case on-board Flash updates can be managed faster than using serial upload (See 1)

- Change to the host directory, containing the kernel.bin and the jffs2.img files.
- The command line
  mtdw kernel.bin /dev/mtd0
  updates the kernel in the on board Flash.
- The command line
  m*tdw jffs2.img /dev/mtd1*
  updates the jffs2 file system in the on board Flash.
- After reset the new kernel and file system get valid.

9.3. *Changing the Ethernet IP Address Permanently*

The IP address can be set using the *ipconfig* command. However, this is temporary only and lost after board reset. To set it permanently, the kernel settings must be changed, the kernel must be recompiled and the kernel + file system must be reflashed.

- The file *uClinux-dist\vendors\NEC\SOClite+\makefile* contains the settings to be changed:

CONFIG_NET_ADDR = 192.168.25.10	-> IP addres
CONFIG_NET_GATEWAY = 192.168.25.254	-> Gateway
CONFIG_NET_NETMASK = 192.0.0.0	-> Sub-Net mask
CONFIG_HOSTNAME = soc	-> Host name

- After the change, follow chapter 1, steps 4 to 7 to recompile the kernel and file system
- Follow chapter 9.2 to update the kernel and file system in the on-board Flash

[MEMO]

A. Bibliography

[1] NEC Corporation, "System-on-Chip Lite+ User's Manual", A17158EE2V0UM00, NEC Corporation, April 2005

[2] Advanced Micro Devices Inc., "Am29DL32xG", Data Sheet, AMD Inc.

[3] Elpida Memory Inc., "EDS2516APTA", Data Sheet, Elpida Memory, Inc., February 2003

[4] Cadence Design Foundry, "Linux API Driver for Ethernet 10/100 MAC", Technical Data Sheet, Cadence (UK), February 2003

[5] ARM Ltd., "ARM Architecture Reference Manual", Manual, ARM Limited.

[6] Embedded Performance Inc., "JEENI Intelligent Debug Probe", http://www.epitools.com/products/jeeni.php.

[7] Alessandro Rubini, Jonathan Corbet, "Linux Device Drivers, 2nd Edition", O'Reilly 2001, http://http://www.oreilly.com/catalog/linuxdrive2/.

[8] Russel King et al., "The ARM Linux Project", http://www.arm.linux.org.uk/.

[9] Erik Andersen, "BusyBox: The Swiss Army Knife of Embedded Linux", http://www.busybox.net/index.html.

[10] Ralf S. Engelschall, "SMTPclient - simple SMTP client", http://www.engelschall.com/sw/smtpclient/.

[11] Greg Ungerer, "Using Flash Memory with uClinux", http://www.cyberguard.com/snapgear/tb20020917.html.

[12] Kimmo Nikkanen, "uClinux As An Embedded Solution", Turku Polytechnic, 2003.

[13] Various (the Open Source community), "READMEs, FAQs, HOWTOs and man pages".

B. Revision History

Revision	Issue Date	Changes
1.0	25.08.2005	Initial release