Customizing Platform Code in Digi Embedded Linux

90001228_B

1 Document History

Date	Version	Change Description
05/28/2009	A	Initial entry/outline
01/05/2011	90001228_A	Transferred document to Digi US repository

2 Table of Contents

3 Introduction

Digi Embedded Linux platforms support different hardware devices and interfaces. To support them in the Linux kernel, providing only the device driver is usually not enough. Some devices need information about the hardware, such as the interrupts or GPIOs they will use, the clock source, initial configuration data, etc. This information is given in the platform code.

The platform code of Digi platforms is customized to match the specific hardware of Digi Development Kits. In practice, however, a user would prefer to add their own hardware (like an I2C or SPI device) to their final base board and control it using the kernel API.

This Application Note helps you understand and customize platform code to add support for custom hardware devices (like I2C and SPI devices) in Digi Embedded Linux.

4 Terminology

Some terms have multiple meanings and overlap depending on the context:

- ***architecture***: processor architecture, such as ARM, MIPS, x86, etc.
- ***platform***: family inside one architecture or group of CPUs with common stuff.
- ***sub-architecture***: same as platform.
- ***machine***: a specific hardware implementation of a platform.

The term *platform* also refers to the physical hardware formed by the CPU module and development board.

In many sources, the terms *machine* and *platform* are used with the same meaning.

5 Understanding the platform code

The Linux kernel can be cross-compiled to different CPU architectures (x86, ARM, MIPS, etc.). Architecture dependent code is placed in the ***arch/*** folder within the Linux kernel source tree. Within this folder, there is a subfolder per architecture, for example:

arch/alpha/	*For ALPHA processors*
arch/arm/	*For ARM processors*
arch/mips/	*For MIPS processors*
arch/x86/	*For x86 processors*
…	

Within each of these architecture subfolders, there are the specific processors or families of processors (known as ***platforms***). Platform-specific code is stored within these subfolders, for example:

arch/arm/mach-at91/	*For Atmel AT91 family of processors*
arch/arm/mach-ns9xxx/	*For NetSilicon NS9xxx family of processors*
arch/arm/mach-s3c2443/	*For Samsung S3C2443 processor*
…	

The selection of the ***System Type*** in the Linux kernel configuration determines the platform code to be compiled.

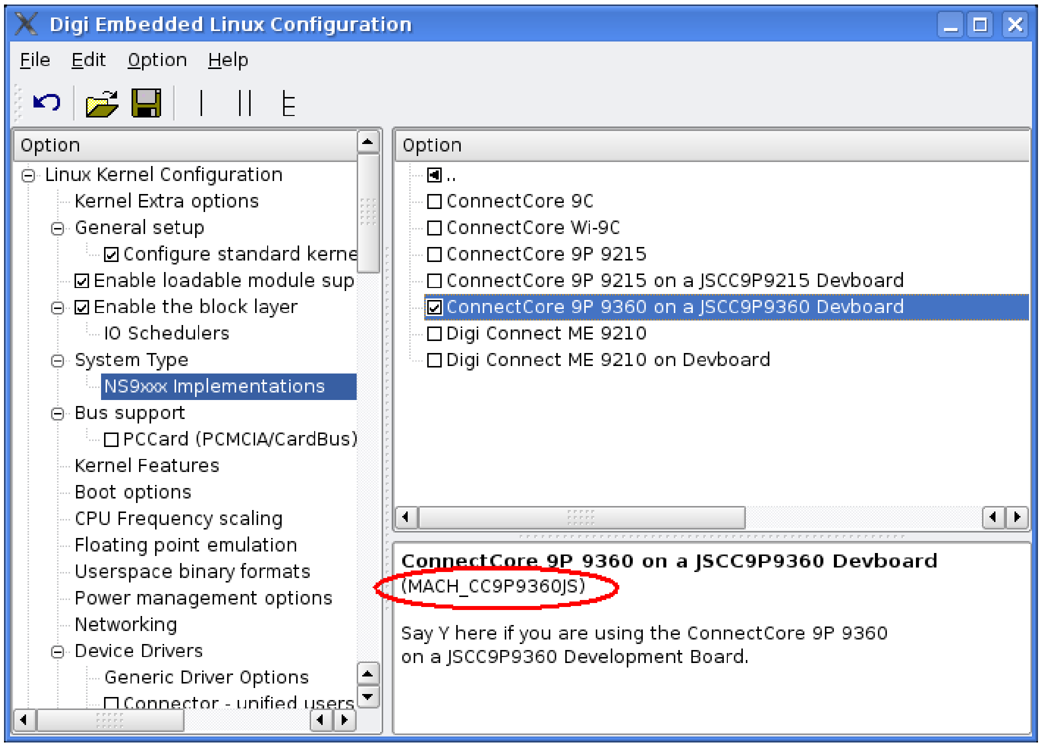
When the platform code comprises more than one processor or hardware board, additional options are given in the kernel configuration allowing you to select the specific processor and hardware platform for which to compile the kernel. This is known as a ***machine***. The selection of a machine basically defines a configuration variable in the form CONFIG_MACH_*PLATFORMNAME*=y, which tells the Makefile in the platform code folder to compile the files containing the code for that specific machine.

5.1 A real example

Let's see a real example. The folder *arch/arm/mach-ns9xxx/* comprises the platform code for several processors (NS9360, NS9215, NS9210). Not all the files in this folder need to be compiled. Selecting the files to be compiled is done within the ***Makefile*** of this folder and is dependent upon configuration variables, for example:

```
obj-$(CONFIG_MACH_CC9P9360JS) += mach-cc9p9360js.o
```

In this excerpt of the Makefile we can see that the code in file *mach-cc9p9360js.c* will only be compiled if the variable *CONFIG_MACH_CC9P9360JS* is set to ***y***. This variable is set by the ***Kconfig*** file of this folder, if the *ConnectCore 9P 9360* platform on a *JSCC9P9360* development board is selected in the kernel configuration tool.



```
Linux Kernel Configuration
  Kernel Extra options
  General setup
    [✓] Configure standard kerne
  [✓] Enable loadable module sup
  [✓] Enable the block layer
    IO Schedulers
  System Type
    NS9xxx Implementations
  Bus support
    [ ] PCCard (PCMCIA/CardBus)
  Kernel Features
  Boot options
  CPU Frequency scaling
  Floating point emulation
  Userspace binary formats
  Power management options
  Networking
  Device Drivers
    Generic Driver Options
    [ ] Connector - unified users
```

```
..
[ ] ConnectCore 9C
[ ] ConnectCore Wi-9C
[ ] ConnectCore 9P 9215
[ ] ConnectCore 9P 9215 on a JSCC9P9215 Devboard
[✓] ConnectCore 9P 9360 on a JSCC9P9360 Devboard
[ ] Digi Connect ME 9210
[ ] Digi Connect ME 9210 on Devboard
```

ConnectCore 9P 9360 on a JSCC9P9360 Devboard
(MACH_CC9P9360JS)

Say Y here if you are using the ConnectCore 9P 9360
on a JSCC9P9360 Development Board.

The code in the file *mach-cc9p9360js.c* is specific to the ConnectCore™ 9P 9360 (CC9P9360) module running on the Digi JSCC9P9360 development board. By selecting this machine, the kernel will add support for the specific hardware contained in that module and development board.

5.2 What is in the platform code?

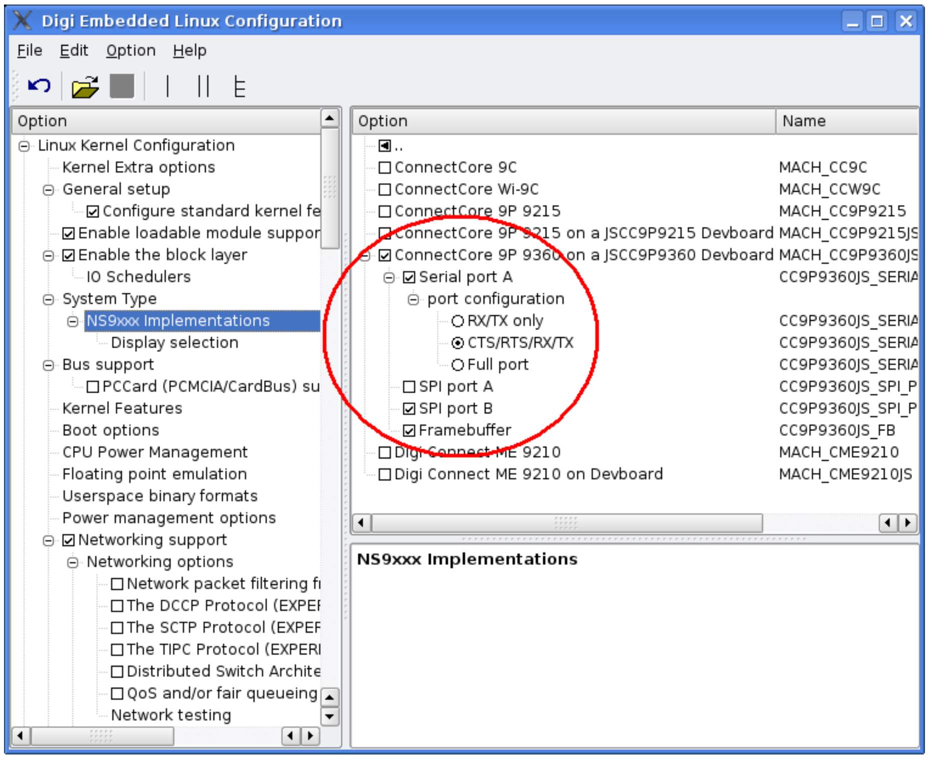
Now, what kind of code does a platform file contain?

The drivers for different hardware devices are stored under the ***drivers/*** folder of the kernel source tree. Device drivers for the Ethernet interface, Serial, SPI controller, I2C controller, Flash, Wireless, etc. are stored here. However, most of these drivers need some information that is specific to the hardware platform; for example, the interrupt they will use, the physical memory address they are mapped to, the GPIOs they will reserve and use, the clock source, initial configuration data, etc.

This is the kind of information that is stored in the platform code and is passed to the device drivers by means of *platform devices*, which are a special kernel structure.

5.3 Can platform code be customized?

In general, the platform folder doesn't need any modification because it contains support for most of the hardware present in Digi modules and development boards. Digi platforms are also prepared to offer users some customizations within the kernel configuration tool itself. These customizations can include whether or not to enable the frame buffer support or the touch screen device, or how many serial ports will be available and how many lines will they use.



```
Option
⊖ Linux Kernel Configuration
    Kernel Extra options
  ⊖ General setup
      ☑ Configure standard kernel fe
  ☑ Enable loadable module suppor
  ⊖ ☑ Enable the block layer
      IO Schedulers
  ⊖ System Type
    ⊖ NS9xxx Implementations
        Display selection
  ⊖ Bus support
      ☐ PCCard (PCMCIA/CardBus) su
    Kernel Features
    Boot options
    CPU Power Management
    Floating point emulation
    Userspace binary formats
    Power management options
  ⊖ ☑ Networking support
    ⊖ Networking options
        ☐ Network packet filtering fi
        ☐ The DCCP Protocol (EXPEF
        ☐ The SCTP Protocol (EXPEF
        ☐ The TIPC Protocol (EXPERI
        ☐ Distributed Switch Archite
        ☐ QoS and/or fair queueing
        Network testing
```

Option	Name
..	
☐ ConnectCore 9C	MACH_CC9C
☐ ConnectCore Wi-9C	MACH_CCW9C
☐ ConnectCore 9P 9215	MACH_CC9P9215
☐ ConnectCore 9P 9215 on a JSCC9P9215 Devboard	MACH_CC9P9215JS
⊖ ☑ ConnectCore 9P 9360 on a JSCC9P9360 Devboard	MACH_CC9P9360JS
⊖ ☑ Serial port A	CC9P9360JS_SERIA
⊖ port configuration	
○ RX/TX only	CC9P9360JS_SERIA
● CTS/RTS/RX/TX	CC9P9360JS_SERIA
○ Full port	CC9P9360JS_SERIA
☐ SPI port A	CC9P9360JS_SPI_P
☑ SPI port B	CC9P9360JS_SPI_P
☑ Framebuffer	CC9P9360JS_FB
☐ Digi Connect ME 9210	MACH_CME9210
☐ Digi Connect ME 9210 on Devboard	MACH_CME9210JS

NS9xxx Implementations

However, Digi platforms don't know about any new or different devices that may be present in your customized hardware. For this reason, it is necessary to create new platform code that covers your new devices and hardware modifications.

6 Use of a custom machine

Using the standard platform code for Digi embedded modules is fine for developing and testing on Digi development boards. However, users may eventually want to compile code for their own hardware board, where different hardware may be present.

This guide presents two ways of adding support to your specific hardware:

- The formal way
- The informal way

Note: The quickest way to add your specific platform code is to directly modify the source code of the Digi platform you are working with. If you are working with a ConnectCore 9P 9360 on a JumpStart development board, that would be the file *arch/arm/mach-cc9p9360js.c* and maybe some additional files which contain related functions.

However, this method is not recommended as it will destroy the platform code for your Development Kit. If you choose to implement this method, remember to do a backup before making any modifications.

6.1 The formal way

ARM platforms are identified in the kernel tree by a machine ID. Machine IDs are allocated by the kernel maintainer in order to keep the large number of ARM platform variants manageable in the source trees. The formal way of adding support for your custom machine requires that you register a new ID for your custom hardware platform.

The new ID also requires you to make some U-Boot adjustments.

Normally, you would only register a new machine ID if the changes in the platform code are so extensive that it makes sense to differentiate the new platform code from the original machine in the kernel sources. Additional information about registering a new machine can be found in *Appendix A: Create a new machine*.

6.2 The informal way

If you want to avoid the overhead of registering a new machine ID into the kernel, you can reuse the ID of the platform which most resembles your hardware. This is the recommended method.

6.2.1 Adding the platform to the Kconfig

Edit the file arch/arm/mach-*similarplatformsuffix*/Kconfig and add an entry to select your platform. For example, if you are basing your new platform on the *ns9xxx* platform, you would edit the file *arch/arm/mach-ns9xxx/Kconfig* with:

```
config MACH_YOURPLATFORM
    bool " CC9P9360 on my super new hardware"
    select MODULE_CC9P9360
    help
      Say Y here if you are using the ConnectCore 9P 9360
      on my Super new hardware board.
```

Note that the third line in the Kconfig script eventually sets the variable CONFIG_MODULE_CC9P9360 which is available in the C code. This variable will force the compilation of the code specific for the CC9P9360 module. You want to leave such a line if you are using that module in your custom hardware.

Similarly, you can add other lines to define any custom variables that you need. These variables will be accessible from the C code by appending the “CONFIG_” prefix to them.

This will result in your custom hardware being available for selection on the kernel configuration tool.

6.2.2 Creating the platform code

Create a new file *arch/arm/mach-yourplatformsuffix/mach-yourplatform.c* based on the platform code that most resembles your hardware. In our example, we will create a new platform *arch/arm/mach-ns9xxx/mach-yourplatform.c* which is a copy of *arch/arm/mach-ns9xxx/mach-cc9p9360js.c*.

The following code needs to be modified to match your platform:

```
MACHINE_START(YOURPLATFORM, "ConnectCore 9P 9360 on my hardware")
    .map_io = ns9360_map_io,
    .init_irq = ns9xxx_init_irq,
    .init_machine = mach_yourplatform_init_machine,
    .timer = &ns9360_timer,
    .boot_params = 0x100,
MACHINE_END
```

Then provide the initialization code for your platform, in the function *mach_yourplatform_init_machine()*. Additional information about customizing the platform code is given later in this document.

Note the CC9P9360 macro is used in the above example because we don’t have registered macros for our custom machine.

6.2.3 Adding the platform to the Makefile

In order for your platform code to be compiled, you need to add a rule to the file *arch/arm/mach-yourplatformsuffix/Makefile*. This is normally a line, based on your platform config macro, that tells the Makefile which object files it needs to compile.

```
obj-$(CONFIG_MACH_YOURPLATFORM) += mach-yourplatform.o
```

6.2.4 Adding a new machine type

We will add a new machine type without registering it upstream into l*inux/arch/arm/tools/mach-types* with a line similar to:

```
#
# machine_is_xxx   CONFIG_xxxx        MACH_TYPE_xxx  number
#
yourplatform       MACH_YOURPLATFORM  YOURPLATFORM   YOURID
```

Make sure that you don't compile two files that have the MACHINE_START code for the same ID. In the example above we reused the ID *CC9P9360JS* and cannot therefore compile the file *mach-cc9p9360js.c*, which implements the MACHINE_START code for this same ID.

You will also need to configure U-Boot to use the new machine ID as explained in Appendix A, section 8.5.

7 Customizing the platform code

As previously mentioned, platform code contains specific information about the devices present in the hardware platform, such as the interrupt they will use, the physical memory address they are mapped in, the GPIOs they will reserve and use, the clock source, initial configuration data, etc.

Most of the code in the platform folder doesn't require any modification since Digi modules contain certain hardware that will always be present and cannot be customized; such as Flash memory, the Ethernet controller, timers and interrupts, the DMA controller, etc.

The final carrier board design for the Digi module may, however, include additional hardware that is not present on the Digi development board, and could also not provide hardware components that are present on the standard Digi development board.

7.1 Removing unneeded devices

Digi's JumpStart boards may contain hardware devices or circuitry for interfaces like:

- Display
- Serial ports
- External I2C RTC chip
- ADC
- Watchdog
- SPI touch screen

The initialization code function *mach_digiplatform_init_machine()* will call other functions to initialize the corresponding devices which are present on the JumpStart board.

Usually the supported devices are included, or not, depending on a configuration variable of the Kconfig file. It is therefore easy to include them (or not) in the kernel configuration. For example, you may find code like the following:

```
static void __init mach_cc9p9360js_init_machine(void)
{
...
    /* Framebuffer */
#if defined(CONFIG_CC9P9360JS_FB)
    ns9xxx_add_device_cc9p9360_fb(18);
#endif
    /* Watchdog timer */
    ns9xxx_add_device_ns9360_wdt();
...
```

In this example, the frame buffer is only included if previously enabled in the kernel configuration (the Kconfig defines variable CONFIG_CC9P9360JS_FB if the frame buffer component is enabled).

Although the watchdog seems as though it is not protected by a similar mechanism, the function *ns9xxx_add_device_ns9360_wdt()* itself is actually defined depending on a similar variable.

Remember, you can always comment or remove the code of the devices that don't exist in your hardware platform.

7.2 Memory mapped hardware

If your custom design adds new hardware to the memory map, for example using a free external chip select of the module, you will need to create the code to reserve the I/O memory space, interrupts, GPIOs, or any additional resource that the new device may need.

7.2.1 Example: charlcd

You can use the code of the *charlcd* driver module, at */usr/local/DigiEL-X.Y/modules/charlcd/*, as an example of a memory mapped device. This is a simple driver for controlling a 16 character x 2 line LCD display. The display is thought to be connected to the peripheral application header connector of a Digi JumpStart development board and use the following lines:

- One external chip select
- Two address lines
- Eight data lines
- One GPIO (for backlight control)

Schematics are given as a reference, and a README file explains how to load and use the driver. The source code has been divided in two files:

- ***charlcd.c***: contains the code for controlling the LCD display. This code is common to all hardware platforms.
- ***lcd_plat.c***: contains the platform code, i.e. the code which differs depending on the platform the LCD is connected to (configuration of the chip select, number of GPIOs, etc.).

The driver code has been designed to exemplify the use of platform devices. In real development, the code in *lcd_plat.c* should go into your platform code *mach-yourplatform.c* with the rest of your devices, instead of in a separate file.

The way it works is the platform code *lcd_plat.c* registers a platform device to be controlled by a driver with the name *charlcd*. On the other hand, the driver code *charlcd.c* registers a platform driver with the name *charlcd*. When this happens, the kernel detects that there is a registered device for this driver and calls the *probe()* function of the driver with the information of the device, which contains the platform-dependent data.

7.3 Bus connected devices

Devices that are connected to a bus constitute a special case. These are, for example, SPI and I2C devices.

Digi module processors contain SPI and I2C controllers, allowing you to connect SPI and I2C devices to them. These bus connected devices are not mapped in physical memory; they are simply connected to the corresponding bus. However, they may need to reserve and use some of the processor resources, like an interrupt signal or a GPIO. SPI and I2C devices need to be registered using the bus API so that the master controller can manage them.

7.3.1 I2C devices

In the platform code that we have taken as reference, we can see the following code regarding I2C master controller and I2C devices:

```
static void __init mach_cc9p9360js_init_machine(void)
{
...
    /* I2C controller */
    ns9xxx_add_device_cc9p9360_i2c();

    /* I2C devices */
    i2c_register_board_info(0,
            i2c_devices,
            ARRAY_SIZE(i2c_devices));
...
```

The first line adds the I2C master controller of the processor. The second line registers the different I2C devices connected to the bus using the *i2c_register_board_info()* function which is declared in *include/linux/i2c.h*. Its arguments are:

- The bus number: If there is only one I2C master controller, this will be 0. If there are two, it can be 0 or 1, and so on.
- An array of struct i2c_board_info, containing the I2C devices connected to the bus
- The number of elements of the given array

Going back to our code, we are passing the array *i2c_devices* which is defined as follows for the CC9P9360JS platform (for clarity, we have omitted several #ifdef clauses that wrap the code):

```
/* I2C devices */
static struct pca953x_platform_data pca9554_data = {
    .gpio_base = 108,
};

static struct i2c_board_info i2c_devices[] __initdata = {
    {
        I2C_BOARD_INFO("pca9554", 0x20),
        .platform_data = &pca9554_data,
    },
    {
        I2C_BOARD_INFO("ds1337", 0x68),
        .irq = IRQ_NS9XXX_EXT0,
    },
};
```

In this example, there are two I2C devices defined:

- A PCA9554 I/O expander at I2C address 0x20. This is a chip that can be found in the CC9P9360 JumpStart board, connected to the I2C bus.
- A DS1337 Real Time Clock at address 0x68. This is a chip that can be found in the CC9P9360 module itself, connected to the I2C bus.

Each device in the array must be initialized using the I2C_BOARD_INFO macro, which assigns the device type (which eventually links to the driver) and the I2C address for the device. Then, additional data can optionally be provided such as an interrupt to be used, or specific platform data for the device (in the example, a struct containing the GPIO number specifying where to start mapping the expander GPIOs).

To see the complete list of parameters, refer to the *i2c_board_info* struct, defined at *include/linux/i2c.h*.

Note that the device type passed in the I2C_BOARD_INFO macro, cannot be an arbitrary string. The I2C device driver registers its name in an i2c_device_id struct. This is the name that must be passed to the macro to link the device to the driver.

You can remove the I2C devices your platform is not using and add new entries for any new I2C device of your hardware.

7.3.2 SPI devices

In the platform code that we have taken as reference, we can see the following code regarding the SPI master controller and SPI devices:

```
static void __init mach_cc9p9360js_init_machine(void)
{
...
    /* SPI */
#if defined(CONFIG_CC9P9360JS_SPI_PORTA)
    ns9xxx_add_device_cc9p9360_spi_porta();
#endif
#if defined(CONFIG_CC9P9360JS_SPI_PORTB)
    ns9xxx_add_device_cc9p9360_spi_portb();
#endif
#if defined(CONFIG_CC9P9360JS_SPI_PORTC)
    ns9xxx_add_device_cc9p9360_spi_portc();
#endif
#if defined(CONFIG_CC9P9360JS_SPI_PORTD)
    ns9xxx_add_device_cc9p9360_spi_portd();
#endif

...

    /* SPI devices */
    spi_register_board_info(spi_devices,
                ARRAY_SIZE(spi_devices));
```

The NS9360 processor is special because it doesn’t have a dedicated SPI controller. Instead, each of its serial ports can be configured either as UARTs or SPI controllers. For this reason, the first lines add the SPI controller of each port, depending on the value of a configuration variable. This means that we can have up to four SPI master controllers in the system.

The line at the end registers the different SPI devices connected to the bus using the *spi_register_board_info()* function which is declared in *include/linux/spi/spi.h*. Its arguments are:

- An array of *struct spi_board_info*, containing all the SPI devices
- The number of elements of the given array

Going back to our code, we are passing the array *spi_devices* which is defined as follows for the CC9P9360JS platform (for clarity, we have omitted several #ifdef clauses that wrap the code):

```
#define CC9P9360JS_TOUCH                  \
    {                           \
        .modalias  = "ads7846",           \
        .max_speed_hz  = 200000,          \
        .irq       = IRQ_NS9XXX_EXT1,     \
        .bus_num        = 0,              \
        .chip_select    = 0,              \
        .platform_data = &cc9p9360js_touch_data, \
    },

static struct spi_board_info spi_devices[] __initdata = {
    CC9P9360JS_TOUCH
    /* Add here other SPI devices, if any... */
};
```

In this example, there is only one SPI device defined:

- An ADS7846 touch screen controller

Each device in the array must at least define the following fields of the *spi_board_struct*:

- ***modalias***: The device type (which eventually links to the driver)
- ***bus_num***: This is the ID of the SPI master controller the device is attached to. The ID of SPI controllers is set also in the platform code, where each master driver is added (in the example, the ID used is 0 which means the touch screen device is attached to the SPI port defined with ID 0 in the CC9P9360JS platform code). If we see an excerpt of the file ns9360_devices.c, we'll see that this ID corresponds to SPI port B of the NS9360 processor.

```
static struct platform_device ns9xxx_device_ns9360_spi_porta = {
    .name       = "spi_ns9360",
    .id     = 1,
    .resource   = spi_porta_resources,
    .num_resources = ARRAY_SIZE(spi_porta_resources),
    .dev = {
        .dma_mask = &spi_dmamask,
        .coherent_dma_mask = DMA_BIT_MASK(32),
    },
};

static struct platform_device ns9xxx_device_ns9360_spi_portb = {
    .name       = "spi_ns9360",
    .id     = 0,
    .resource   = spi_portb_resources,
    .num_resources = ARRAY_SIZE(spi_portb_resources),
    .dev = {
        .dma_mask = &spi_dmamask,
        .coherent_dma_mask = DMA_BIT_MASK(32),
    },
};

static struct platform_device ns9xxx_device_ns9360_spi_portc = {
    .name       = "spi_ns9360",
    .id     = 2,
    .resource   = spi_portc_resources,
    .num_resources = ARRAY_SIZE(spi_portc_resources),
    .dev = {
        .dma_mask = &spi_dmamask,
        .coherent_dma_mask = DMA_BIT_MASK(32),
    },
};
```

```
static struct platform_device ns9xxx_device_ns9360_spi_portd = {
    .name      = "spi_ns9360",
    .id    = 3,
    .resource  = spi_portd_resources,
    .num_resources = ARRAY_SIZE(spi_portd_resources),
    .dev = {
        .dma_mask = &spi_dmamask,
        .coherent_dma_mask = DMA_BIT_MASK(32),
    },
};
```

- ***chip_select***: This is the chip select of the SPI master controller the device is attached to. Some SPI master controllers support driving multiple devices with the use of SPI chip select lines. The SPI ports of the NS9360 processor only support one device each, so we used a 0 for this field.

Additional data can optionally be given, like the interrupt, the max clock speed, or specific platform data for the device (in the example, a pointer called *cc9p9360js_touch_data* which points to a struct containing the limit coordinates of the touch screen as well as other information needed by the driver).

For a complete list of parameters, refer to the *spi_board_info* struct defined at *include/linux/spi/spi.h*.

Note that the device type passed in the modalias field, cannot be an arbitrary string. The SPI device driver registers its name in a spi_driver struct. This is the name that must be passed in the modalias field to link the device to the driver.

You can remove the SPI devices your platform is not using and add new entries for any new SPI device of your hardware.

8 Appendix A: Create a new machine

8.1 Registering a Machine ID

You must first register your new machine with the kernel maintainer to get a number for it. This is not actually necessary to begin work, but you'll need to do this eventually so it's best to do it at the beginning and not have to change your machine name or ID later.

Registration of a new platform can be done online at: http://www.arm.linux.org.uk/developer/machines/

The following information needs to be supplied:

- ***Machine name***: This is the long name of your platform and can be any text
- ***Machine type***: This is the short name for your platform (no spaces allowed)
- ***Directory suffix***: This is the suffix name of the platform folder. Although you can provide a new one, it is recommended to use the same name as the platform you are basing your platform on. This will allow you to reuse pre-existing code.
- ***Web site***: The web site of the platform (if any)
- ***Description***: The long description of your platform

For example, if you have created a new hardware board for the ConnectCore 9P 9360:

- Machine name: *CC9P9360 on my super new hardware*
- Machine type: *YOURPLATFORM*
- Directory suffix: *ns9xxx*
- Web site: http://my-super-new-machine.net
- Description: *ConnectCore 9P 9360 running on my brand new hardware platform, which contains these and those devices*

Upon registration you will receive the machine ID and macros for your platform. Then you need to add this information to linux/arch/arm/tools/mach-types with a line like this:

```
#
# machine_is_xxx   CONFIG_xxxx        MACH_TYPE_xxx  number
#
yourplatform       MACH_YOURPLATFORM  YOURPLATFORM   YOURID
```

Or, go to: http://www.arm.linux.org.uk/developer/machines/ where you can download the latest version of the mach-types file.

The above file is needed so that the kernel script *linux/arch/arm/tools/gen-mach-types* can generate *linux/include/asm-arm/machtypes.h* which sets the necessary defines and macros that are used by much of the source to select the appropriate code.

Keep in mind that this file won't be synchronized with the official Linux kernel until the next major kernel version is released, so do not update this file if updating the Linux kernel source tree.

8.2 Creating the platform Kconfig

To allow different options to be selected during the kernel configuration, and also set some constants for the build process, you need to create a file *arch/arm/mach-yourplatformsuffix*/Kconfig and add an entry to select your new platform:

```
config MACH_YOURPLATFORM
    bool " CC9P9360 on my super new hardware"
    select MODULE_CC9P9360
    help
      Say Y here if you are using the ConnectCore 9P 9360
      on my Super new hardware board.
```

Note that the third line in the Kconfig script eventually sets the variable CONFIG_MODULE_CC9P9360 which is available in the C code. This variable will force the compilation of the code specific for the CC9P9360 module. You want to leave such a line if you are using that module in your custom hardware.

Similarly, you can add other lines to define any custom variables that you need. These variables will be accessible from the C code by appending the “CONFIG_” prefix to them.

This will result in your custom hardware being available for selection on the kernel configuration tool.

8.3 Creating the platform code

Create a new file *arch/arm/mach-yourplatformsuffix/mach-yourplatform.c* based on the platform code that most resembles your hardware. In our example, we will create a new platform *arch/arm/mach-ns9xxx/mach-yourplatform.c* which is a copy of *arch/arm/mach-ns9xxx/mach-cc9p9360js.c*.

The following code needs to be modified to match your platform:

```
MACHINE_START(YOURPLATFORM, "ConnectCore 9P 9360 on my hardware")
    .map_io = ns9360_map_io,
    .init_irq = ns9xxx_init_irq,
    .init_machine = mach_yourplatform_init_machine,
    .timer = &ns9360_timer,
    .boot_params = 0x100,
MACHINE_END
```

Then provide the initialization code for your platform, in the function *mach_yourplatform_init_machine()* and add code for the rest of your devices, as explained in this document.

8.4 Adding the platform to the Makefile

In order for your platform code to be compiled, you need to add a rule to the file *arch/arm/mach-yourplatformsuffix/Makefile*. This is normally based on your platform config macro and that tells the Makefile which object files it needs to compile.

```
obj-$(CONFIG_MACH_YOURPLATFORM) += mach-yourplatform.o
```

8.5 Configure U-Boot

The U-Boot boot loader passes the machine ID to the kernel, and this ID must match the one the kernel expects (the ID of the machine it was compiled for), otherwise the kernel will refuse to boot.

The machine ID is hardcoded in the U-Boot image, but can be forced using a variable called *machid* in U-Boot.

In order for your kernel to boot, you must change this variable to contain the ID number of your new machine. For example, if the ID of your machine is 9999:

```
#   setenv machid 9999
#   saveenv
```