AppNote-126

PCI Configuration For BSP Developers

What BSP and Device Driver Developers should know about Wind River's PCI Autoconfiguration Software.

Abstract

The purpose of this document is to describe advanced features of PCI configuration and new features added in Tornado 2.2. This document is intended as a replacement for WTN-49 "The Peripheral Component Interconnect (PCI) Bus and vxWorks". Some familiarity with PCI is assumed. It is intended for BSP developers, device driver writers, and others who need to know about hardware configuration.

Version 1.0 April 2002

Overview

Wind River provides support for the Peripheral Component Interconnect (PCI) bus in many of its Board Support Packages (BSPs). There are two methods of configuring the PCI bus: manual configuration or automatic configuration. With manual configuration, the BSP developer determines in advance what hardware will be present on the PCI bus and uses functions from **pciConfigLib.c** to configure the buses and devices. With automatic configuration, the BSP developer determines overall configuration variables and calls functions from **pciAutoConfigLib.c** to configure the bus automatically. In the second case, **pciAutoConfigLib.c** functions will call the **pciConfigLib.c** functions as necessary. When **pciAutoConfigLib.c** is used, knowledge of the specific devices on the PCI bus is not required for PCI configuration. This document focuses mostly on the use of **pciAutoConfigLib.c**.

Resources

The PCI bus provides three types of address space: I/O, memory, and configuration. Each device is mapped to memory and/or I/O space through Base Address Registers (BARs) located in configuration space. This eliminates the need for hardware jumpers to determine the addresses for the device's registers. The configuration of the PCI bus is almost completely controlled by software registers in configuration space.

Therefore, each PCI device must be configured before it can be used. This means that its memory or I/O address must be assigned and the device must be enabled to respond to normal PCI transactions.

Before configuring the PCI addresses, the address ranges allocated to the different types of address space must be determined. The configuration space is determined by the PCI spec, so the BSP developer does not need to worry about it. However, address ranges for I/O and memory spaces need to be determined. For I/O and memory space, it is possible to have ranges specified by different address sizes. For I/O space, you can access a device using 16-bit addresses or 32-bit addresses. For memory space, 32-bit space is available. Although 64-bit addressing for memory space is defined by the PCI spec, **pciAutoConfigLib.c** and **pciConfigLib.c** do not currently support it. In addition to address size, memory space can also be classified as prefetchable or non-prefetchable. So there are a total of four address ranges which need to be determined: 16-bit I/O space (io16), 32-bit I/O space (io32), 32-bit prefetchable memory space (mem32), and 32-bit non-prefetchable memory space (memIo32).

For each of these spaces, the BSP developer needs to determine a starting address and a maximum size. The size chosen must be large enough to support all devices which might be put on the bus. However, since PCI address spaces are accessed as memory, the MMU needs to be configured to map the addresses used. This has two consequences.

First, before the device's registers can be accessed, the MMU needs to be configured and initialized for the address ranges in use in PCI space. This can be done either before or after

PCI configuration is accomplished. In either case, the device drivers cannot initialize the devices until after both PCI and MMU initializations are performed.

Second, for some processor architectures and configurations, using PCI memory and/or I/O space may require that the MMU's Page Table Entries (PTEs) be placed in the system's main memory. In this case, allocating large amounts of memory and/or I/O space to PCI will thus consume main memory and reduce the amount of memory which is available for applications. For a system with a limited amount of main memory, this can be significant, so it may be wise to restrict the amount of PCI space which can be allocated.

PCI Interrupts

The PCI specification does not address how interrupt signals are routed to the interrupt controller device for a motherboard bus. Each device has four interrupt pins available. They are named A, B, C, and D. Each single interrupt PCI device is required to always use Int Pin A to generate an interrupt. Devices with multiple functions can assign one interrupt pin per function. If a device implements all eight possible sub-functions, there might be two interrupt sources on each interrupt pin, though other configurations are possible as well. A PCI interrupt handling system needs to be able to call several Interrupt Service Routines (ISRs) for each generated interrupt. The normal operation is to call all ISRs attached to the interrupt pin each time an interrupt occurs from that pin. Each handler is responsible for checking that its associated device is actually generating an interrupt. If it isn't, the handler returns immediately so that the next ISR can be called.

The module pciIntLib.c provides for multiple interrupt handlers to be attached to a single interrupt line. This is done by installing a special handler that calls each of the ISRs from a linked list. The **pciIntConnect()** and **pciIntDisconnect()** functions simply add or delete handlers from the linked list.

Typical Calling Sequence

When vxWorks is started, the first use of any device occurs just after the call to **sysHwInit2()**. However, the MMU is initialized and activated between the calls to **sysHwInit()** and **sysHwInit2()**. Because of this, many BSP developers choose to configure the PCI bus early in sysHwInit2(), so that devices can be accessed immediately after PCI configuration. However, it is also possible to call it near the end of **sysHwInit()**, and there are some benefits to doing so.

Typically, the BSP directory contains a file called **sysBusPci.c**. This file contains the BSP-specific functions which are necessary for PCI configuration. The primary function is **sysPciAutoConfig()**, which is called from **sysHwInit()** or **sysHwInit2()**. The function **sysPciAutoConfig()** defines the configuration policy and determines what resources are to be available for PCI configuration, and then calls the function **pciAutoConfig()** or **pciAutoCfg()**.

When **pciAutoConfig()** is called, it goes out to PCI configuration space and performs PCI configuration for each function on the bus. In Tornado 2.2, a new API is available to make the PCI configuration software easier to maintain and update. With the new API, the actual configuration is performed by a function called **pciAutoCfg()**. The old **pciAutoConfig()** function is kept for backward compatibility, but some new features will not be available if this interface is used. Except as otherwise noted, comments about **pciAutoConfig()** in this document also apply to functions which are part of the new API.

The code within **pciAutoConfigLib.c** does a multi-pass configuration. The first pass disables each device, assigns bus numbers, and builds a list of the devices on the bus. A second pass assigns BAR addresses and does other configuration for the bridges and devices. Note that pciAutoConfigLib configures only the bus-specific parts of initialization. Setting the device's registers and software configuration is done by the BSP and device drivers at a later time.

As specified above, **sysPciAutoConfig()** is typically called from within **sysHwInit()** before most other hardware is initialized. This includes the console, so serial output can not be printed during pci configuration.

After PCI configuration is done, the BSP will find all instances of each device type, using the functions **pciFindDevice()** and/or **pciFindClass()**, configure the device, and hand it over to the appropriate driver to control.

pciAutoCfgCtl()

There are several methods for the BSP and device drivers to get information to and from the PCI configuration software. There are a number of configuration options which can be specified using callback mechanisms, or hooks. There are also several ways to get information back from pciAutoConfigLib.c for further hardware and system customizations by the BSP. These include the addresses to use for different PCI spaces, as described in the pciAutoConfigLib documentation, and several callback functions. Each of these can be set by calls to **pciAutoCfgCtl()**, as described in the documentation for that routine.

Historically, the primary way to provide information to the PCI configuration software was a structure of type **PCI_SYSTEM**. This structure provides fields which define options for PCI configuration and determine resources which are dedicated to the PCI bus. Starting with Tornado 2.2, the newer **pciAutoCfgCtl()** interface can be used to set the same values. When using the newer **pciAutoCfgCtl()** interface, each of the values in the **PCI_SYSTEM** structure can be set by individual calls to the function **pciAutoCfgCtl()**. However, some of the new functionality cannot be configured within the **PCI_SYSTEM** structure. For new development, the **pciAutoCfg()** and **pciAutoCfgCtl()** interface should be used exclusively.

The individual options available for PCI configuration, and the meaning of each option, are described in the documentation for **pciAutoCfgCtl()**. The following sections add additional detail to that documentation.

Hooks

The specific hooks available are:

includeRtn
intAssignRtn
bridgePreConfigInit
bridgePostConfigInit
pciRollcallRtn
pciLogMsgFunc (Tornado 2.2 and later)
pciMaxLatFunc (Tornado 2.2 and later)
functionPreConfigRtn (Tornado 2.2 and later)

The discussion below builds on the descriptions of the callback functions in the pciAutoConfigLib documentation.

A pciRollcallRtn

Sometimes, it can be a while before some devices are ready to be configured when the power is first applied. If PCI configuration is attempted before some devices are ready, those devices will not be configured. To handle this situation, pciAutoConfigLib allows the BSP developer to include a rollcall routine to delay PCI configuration. The specified routine is called repeatedly until it returns TRUE. Each time after the first call, pciAutoConfigLib builds a list of PCI functions present, which **pciRollcallRtn** can check to see if the required devices are present.

Although the above description is the intended way for **pciRollcallRtn** to be used, the BSP developer can write **pciRollcallRtn** in any way that works, provided that it returns TRUE only after the PCI bus is ready to be configured. The algorithm can be to check specific devices, to simply count the number of devices present, a simple timeout, or whatever other algorithm makes sense. The following code presents an example of using the rollcall routine to insure that a list of specific devices is found, before PCI configuration is attempted.

```
/* PCI autoconfig roll call support */
/* Roll call list entry structure, list elements specified in "config.h" */

typedef struct _PCI_ROLL_CALL_LIST
    {
    UINT count;
    UINT Dev;
    UINT Vend;
    } PCI_ROLL_CALL_LIST;

LOCAL PCI_ROLL_CALL_LIST rollCall[] =
    {
    PCI_ROLL_CALL_LIST_ENTRIES
    { 0xffff, 0xffff, 0xffff }  /* Required entry: marks end of list */
    };

/*******************************************************************************
*
* sysPciRollCallRtn - Check "roll call" list against list of PCI devices found
*
* This function checks if the number of devices actually found during
* the 1st pass of PCI autoconfiguration (bus enumeration process)
* passes the "roll call" test.  That is, for each entry in the roll call
* list (consisting of a count and device/vendor ID), a check is made to
* insure that at least the specified minimum number of devices has
* actually been discovered.  If the roll call passes, the function returns
* TRUE.  If the roll call fails and the time duration in seconds represented
* by ROLL_CALL_MAX_DURATION has not elapsed, the function will wait 1
* second and return FALSE.  If the roll call fails and the time duration in
* seconds represented by ROLL_CALL_MAX_DURATION has elapsed, the function
* will return TRUE.
*
* Note that this uses the function sysMsDelay(), which must be supplied by
```

```
* the BSP.
*
* RETURNS: TRUE if roll call test passes or timeout, FALSE otherwise.
*/

LOCAL STATUS sysPciRollcallRtn
    (
    )
    {
    STATUS  rollCallPass;       /* Flag indicating pass or fail */
    int     rollIndex;
    UINT    bus;
    UINT    dev;
    UINT    func;
    int     count;
    static  int secDelay = -1;

    if (secDelay == -1)
       secDelay = ROLL_CALL_MAX_DURATION;

    rollCallPass = TRUE;        /* Default = "passed" */

    rollIndex = 0;

    while (secDelay >= 0)
        {

        if (rollCall[rollIndex].Vend == 0xffff)
            break;              /* End of roll call list, we're done */

        count = 0;

        while (pciFindDevice(rollCall[rollIndex].Vend, rollCall[rollIndex].Dev,
                          count, &bus, &dev, &func) == OK)
           count++;

        if (count < rollCall[rollIndex].count)
            {
           secDelay--;
           if (secDelay < 0)
               {
               rollCallPass = TRUE;      /* Timeout, say we passed */
               break;
               }
           else
               {
                rollCallPass = FALSE;       /* Roll call - someone is missing */
                sysMsDelay(1000);           /* Delay a second */
                break;
               }
            }

       rollIndex++;
        }

    if (rollCallPass == TRUE)
       secDelay = -1;

    return (rollCallPass);
    }
```

B includeRtn

The functions in pciAutoConfigLib allow the BSP developer to specify whether individual devices should be configured or not. This allows BSP developers to avoid wasting PCI resources for devices which are not supported.

Before configuring any individual device, **pciAutoConfig()** checks to see whether includeRtn was specified. If it is, then the specified routine is called with three arguments. The first is the **PCI_SYSTEM** structure which was passed to **pciAutoConfig()**. The second is a **PCI_LOC** structure which describes the device, including bus, device, function, attribute, and offset. The third parameter is a 32-bit value containing the device ID and vendor ID. To avoid configuring the device, the includeRtn must return **ERROR**.

The sample code below explicitly lists the known devices to include or exclude.

```
/******************************************************************************
*
* sysPciAutoConfigInclude - Determine if function is to be autoConfigured
*
* This function is called with PCI bus, device, function, and vendor
* information.  It returns an indication of whether or not the particular
* function should be included in the automatic configuration process.
* This capability is useful if it is desired that a particular function
* NOT be automatically configured.  Of course, if the device is not
* included in automatic configuration, it will be unusable unless the
* user's code made provisions to configure the function outside of the
* the automatic process.
*
* RETURNS: OK if function is to be included in automatic configuration,
*          ERROR otherwise.
*/

LOCAL STATUS sysPciAutoConfigInclude
    (
    PCI_SYSTEM *pSys,       /* input: AutoConfig system information */
    PCI_LOC *pciLoc,        /* input: PCI address of this function */
    UINT     devVend        /* input: Device/vendor ID number      */
    )
    {
    BOOL retVal = OK;

    /* If it's the host bridge then exclude it */

    if ((pciLoc->bus == 0) && (pciLoc->device == 0) && (pciLoc->function == 0))
        return ERROR;

    switch(devVend)
       {

       /* EXCLUDED Devices */

       case PCI_ID_IBC:
               retVal = ERROR;
               PCI_AUTO_DEBUG_MSG("sysPciAutoconfigInclude: Excluding IBC\n",
                       0, 0, 0, 0, 0, 0);
               break;

        ....

       /* INCLUDED Devices */

       case PCI_ID_PRI_LAN:
               retVal = OK;
               PCI_AUTO_DEBUG_MSG("sysPciAutoconfigInclude: Including Ethernet\n",
                       0, 0, 0, 0, 0, 0);
               break;


       ....

       default:
               retVal = OK;
               PCI_AUTO_DEBUG_MSG("sysPciAutoconfigInclude: Include unknown device\n",
                       0, 0, 0, 0, 0, 0);
```

```
                break;
            }

        return retVal;
        }
```

The **includeRtn()** can also be used to do configuration for non-standard devices. For example, let's say you are planning to use a processor on a separate card plugged into the PCI bus, for example, a PMC card with a processor and dual-ported memory on it. You know the device/vendor ID (devVend) has a specific device and vendor code, which is available in a preprocessor macro called PCI_ID_MY_PMC_PROCESSOR. Furthermore, let's assume that the amount of memory from that card which is made available to the main processor depends on the final application, which won't be known until a specific driver is loaded. The BSP can configure the processor's memory window to allocate sufficient space for the largest possible application, but only map a smaller area. This will allow the driver to adjust the size of the mapped region on the fly, without worrying about overlapping the address range of other devices or functions on the PCI bus.

To do this, you would include an additional case to the above routine, dealing with PCI_CLASS_PROCESSOR. In this switch statement, you would need to allocate from one of the address ranges, and set the values of the appropriate BARs. The case statement might look something like:

```
            case PCI_ID_MY_PMC_PROCESSOR:
                /* return ERROR to prevent normal configuration */
                retVal = ERROR;
                PCI_AUTO_DEBUG_MSG("sysPciAutoconfigInclude: manual PMC configuration\n",
                               0, 0, 0, 0, 0, 0);
                /* user-supplied routine to fully configure the device */
                myPmcProcessorConfig(pSys, pciLoc->bus, pciLoc->device, pciLoc->function);
                break;
```

Within the function **myPmcProcessorConfig()**, you will need to allocate memory from the **PCI_SYSTEM** structure for use by the dual-ported memory. The following support function demonstrates how to manage the **PCI_SYSTEM** structure fields to allocate space from the non-prefetchable memory space.

```
/***********************************************************************
*
* pciMemIo32Alloc - allocate addresses in non-prefetchable 32-bit memory space
*
* Allocate the specified address range in non-prefetchable 32-bit
* memory space.  This function insures that the address range will be
* aligned correctly.
*
* ERRNO: not set
*
* RETURNS: The address of the beginning of the
*
*/

UINT    pciMemIo32Alloc
    (
    PCI_SYSTEM *        pSystem,
    UINT                size
    )
    {
    int retStat;
    int sizeAdj;
    void * preAllocAddr;

    retStat = pciAutoAddrAlign(pSystem->pciMemIo32,
```

```
                    pSystem->pciMemIo32 + pSystem->pciMemIo32Size,
                    size,
                    (int *)&preAllocAddr);
    if ( retStat == ERROR )
       {
       return(retStat);
       }

    sizeAdj = ((int)preAllocAddr - pSystem->pciMemIo32) + size;
    pSystem->pciMemIo32 += sizeAdj;
    pSystem->pciMemIo32Size -= sizeAdj;
    PCI_LOG_MSG("pciMemIo32Alloc: addr=0x%08x\n",
                (int)preAllocAddr,
                2,3,4,5,6);

    return((UINT)preAllocAddr);
    }
```

NOTE: The PCI spec states that all PCI cards must know the resource requirements at the time the system boots. For this reason, the usage being described here violates the PCI spec, and is not recommended. Also, future versions of the PCI configuration software may not allow modification of the **PCI_SYSTEM** structure as is done in the sample routine **pciMemIo32Alloc()**.

C bridgePreConfigInit

Some devices need to have specific actions performed on them before the PCI configuration can be successfully accomplished. These devices violate the PCI spec, and cannot be handled by normal PCI configuration. The actions cannot be done before **pciAutoConfig()** is called, since the bus numbers have not been set at that time.

Use of devices such as this should be discouraged, however it may be useful to allow software development during hardware design and debug. To allow these devices to be used, pciAutoConfigLib includes a hook which will be run during the second pass, so that the bus numbers have all been set, but no devices on that bus have yet been configured. The **bridgePreConfigInit** hook can be written and installed so that it will perform the actions required by such non-standard devices.

Here is a trivial example of this routine showing the calling sequence.

```
/******************************************************************************
*
* sysPciAutoconfigPreEnumBridgeInit - PCI autoconfig support routine
*
* RETURNS: N/A
*/

void sysPciAutoconfigPreEnumBridgeInit
    (
    PCI_SYSTEM * pSys,              /* PCI_SYSTEM structure pointer */
    PCI_LOC * pLoc,                 /* pointer to function in question */
    UINT devVend                    /* deviceID/vendorID of device */
    )
    {
    return;
    }
```

D bridgePostConfigInit

If a post-enumeration initialization callback function is specified, then during the second pass through the bus it is called after almost all other initialization is done. The functions on this bus have all been initialized and sub-busses have been traversed. The only configuration done after **bridgePostConfigInit()** is for **MAX_LAT** to be set and for the bridge's status bits to be updated. Here is a trivial example of this routine, showing the calling sequence.

```
/*******************************************************************************
*
* sysPciAutoconfigPostEnumBridgeInit - PCI autoconfig support routine
*
* RETURNS: N/A
*/

void sysPciAutoconfigPostEnumBridgeInit
    (
    PCI_SYSTEM * pSys,               /* PCI_SYSTEM structure pointer */
    PCI_LOC * pLoc,                  /* pointer to function in question */
    UINT devVend                     /* deviceID/vendorID of device */
    )
    {
    return;
    }
```

E intAssignRtn

The PCI spec gives the board designer a lot of freedom in how to route interrupts. There are four PCI interrupt lines, called intA, intB, intC, and intD. They can be combined in many different configurations by the hardware. The BSP developer should understand how the interrupts are treated by hardware and write interrupt assignment software to configure interrupts as appropriate.

In the simplest configuration, the board designer can route all PCI interrupts to a single input pin on an interrupt controller, or even route them directly to an external interrupt line on the processor. In this case, the interrupt assignment will be a constant, the same for all PCI devices.

More commonly, the four PCI interrupt lines are routed to four different pins on an interrupt controller. In this case, devices on each of the four different interrupt lines should be assigned a different interrupt line, depending on the interrupt controller.

Another possible interrupt routing scheme when there are multiple levels of PCI busses is for the different interrupts on a given bus to be shifted one place from the next-higher bus. On bus #1 on such a system, intB would be connected to intA on bus #0. Bus #1's intC would be connected to bus #0's intB. Bus #1's intD would be connected to bus #0's intC. And bus #1's intA would be connected to bus #0's intD. A similar scheme could be used for slots on a single bus, instead of for busses.

The point is that hardware design determines how the **intAssignRtn()** will be written.

The following source code is table-driven. The **intLine[]** table lists the interrupt line values for each device on the bus. This was taken from a BSP which did not support PCI to PCI bridges, so all devices on the PCI bus are on bus zero, eliminating any complexity caused by multiple bus levels.

```
/******************************************************************************
*
* sysPciAutoConfigIntAssign - Assign the "interrupt line" value
*
* RETURNS: "interrupt line" value.
*
*/

LOCAL UCHAR sysPciAutoConfigIntAsgn
    (
    PCI_SYSTEM * pSys, /* input: AutoConfig system information */
    PCI_LOC * pFunc,
    UCHAR intPin      /* input: interrupt pin number */
    )
    {
    UCHAR irqValue = 0xff;    /* Calculated value                */


    if (intPin == 0)
        return irqValue;

    irqValue = intLine [(pFunc->device)][(intPin - 1)];

    PCI_AUTO_DEBUG_MSG("intAssign called for device [%d %d %d] IRQ: %d\n",
                pFunc->bus, pFunc->device, pFunc->function,
                irqValue, 0, 0 );

    return (irqValue);
    }
```

New Features in Tornado 2.2

Starting with Tornado 2.2, a new method is available for calling pciAutoConfig(). The function **pciAutoConfig()** is still available, but additional functionality can be achieved by using the new routine **pciAutoCfg()**. BSPs using the old mechanism can easily be converted to use the new API. Instead of simply calling **pciAutoConfig(&sysParams)**, as with the old API, you now would make the following calls:

```
void * pPciCookie;

pPciCookie = pciAutoConfigLibInit();
pciAutoCfgCtl(pPciCookie, PCI_PSYSTEM_STRUCT_COPY, &sysParams);
pciAutoCfg(pPciCookie);
```

By using this API, additional functionality is available. The system can now be configured to use Fast Back To Back transactions. **MAX_LAT** values can now be set on a per-device basis, rather than a single number for all devices on the system, and a message logging function is now available which allows PCI debug messages to be displayed, even if the serial port is not configured at the time that pciAutoConfigLib is used.

A Fast Back To Back

Fast Back To Back transactions are allowed in the PCI spec. This allows the hardware to perform multiple transactions without negotiating for bus control for each transaction. This is only allowed if all devices on a given sub-bus allow Fast Back To Back transaction types.

The Wind River implementation provided with Tornado 2.2 allows Fast Back To Back transactions to be enabled. If all functions on the PCI bus enable Fast Back To Back transactions, then they will be configured to use this mechanism. However, if there is any function on the bus which does not support Fast Back To Back, then all devices on the bus will be configured not to use it.

The default is to disable Fast Back To Back transactions. To enable it, use the new API. After making the call to **pciAutoCfg()**, issue the command:

```
STATUS pciFbbStatus;
pciAutoCfgCtl(pPciCookie, PCI_FBB_ENABLE, &pciFbbStatus);
```

After this call, **pciFbbStatus** will be set to TRUE if all functions on the bus are configured to use Fast Back To Back transactions, or false if they are not.

There are actually several additional **pciAutoCfgCtl()** commands for working with Fast Back To Back. They are: **PCI_FBB_ENABLE**, **PCI_FBB_DISABLE**, **PCI_FBB_UPDATE**, and **PCI_FBB_STATUS_GET**.

PCI_FBB_ENABLE enables Fast Back To Back transactions, checks to see if all functions on the bus support Fast Back To Back, and configures the devices if appropriate. The third argument, pciFbbStatus, is optional. If specified, it will be set to **TRUE** if Fast Back To Back is configured.

PCI_FBB_DISABLE disables Fast Back To Back transactions. The third argument is ignored.

PCI_FBB_UPDATE, like **PCI_FBB_ENABLE**, checks to see if Fast Back To Back is supported by all devices on the bus, and configures the devices to use it, if appropriate. The third argument, **pciFbbStatus**, is optional. If specified, it will be set to **TRUE** if Fast Back To Back is configured. The difference between **PCI_FBB_UPDATE** and **PCI_FBB_ENABLE** is that the enable request will set a flag indicating that the BSP would like to have fast back to back transactions, and the update request will simply check the flag and proceed only if a previous enable request was issued.

PCI_FBB_STATUS_GET requires the third argument. It sets the value of **pciFbbStatus** to **TRUE** if Fast Back To Back is enabled. It does not check whether Fast Back To Back is configured, so if there is a device present which does not support Fast Back To Back, **pciFbbStatus** can be set to **TRUE** even though no Fast Back To Back transactions can occur.

B pciMaxLatFunc

Versions of pciAutoConfigLib prior to Tornado 2.2 allowed the **MAX_LAT** value for all cards to be specified by the BSP developer. However, the PCI spec allows each device to have a different **MAX_LAT** value. The Tornado 2.2 version allows the BSP developer to specify a routine which determines the **MAX_LAT** value for each card individually.

To configure this, you must use the new API, calling **pciAutoCfg()** instead of **pciAutoConfig()**. Before the call to **pciAutoCfg()**, install the hook by issuing the calls:

```
pciAutoCfgCtl(pPciCookie, PCI_MAX_LAT_ARG_SET, pArg);
pciAutoCfgCtl(pPciCookie, PCI_MAX_LAT_FUNC_SET,(PCI_MAX_LAT_FUNC)pciMaxLatRtn);
```

The hook routine will be called with four arguments: bus, device, function, and user-supplied argument. The following sample code illustrates a simple constant assignment to each PCI function except network controllers, which are assigned a different constant.

```
/**************************************************************************
 * pciMaxLatRtn - determine the MAX_LAT value for a particular PCI function
 *
 * This routine determines the MAX_LAT value for devices on the PCI bus.
 * For network devices, one value is used.  For all other devices, a default
 * value is used.  To determine whether the device is a network device,
 * the device is queried for the class code.
 *
 * RETURNS: the 8-bit unsigned MAX_LAT value.
 */

UINT8 pciMaxLatRtn(UINT bus, UINT device, UINT function, void *pArg)
    {
    UINT8 classCode;

    /* find the device class */
    pciConfigInByte (bus, device, function, PCI_CFG_CLASS,
                     &classCode);

    /* check for network controller */
    if (classCode == CLASS_NET_CNTLR)
        {
        /* return value specific to network devices */
        return(PCI_LAT_TIMER_NET);
        }

    /* return default value */
    return(PCI_LAT_TIMER);
    }
```

C pciLogMsgFunc

The default message output from **pciAutoConfigLib.c** uses **logMsg()**, which will print to the console if the console has been configured and if **logMsg()** has been configured, or drop the message otherwise. Often, PCI configuration is done before the console has been configured, so diagnostic and debug information is simply lost.

It is possible to specify a routine to handle message output. Some BSP developers write a polled-mode output function for use before the console is initialized, which may be called something like **kprintf()**. If you have such a routine available, it can be specified as the output routine. Or you can use a function which sets the LEDs to a known pattern and/or saves messages to RAM for later display. Or any other mechanism which seems appropriate can be used. To use this functionality, the new API is required.

To install the function, make a call to **pciAutoCfgCtl()** immediately after the **pciAutoConfigLibInit()** call:

```
pPciCookie = pciAutoConfigLibInit(NULL);
pciAutoCfgCtl(pPciCookie, PCI_MSG_LOG_SET, (PCI_LOGMSG_FUNC)pciLogMsg);
```

The specified function must take the same arguments as **logMsg()**:

```
void pciLogMsg(char *fmt, int a1, int a2, int a3, int a4, int a5, int a6);
```

Note that this option is only effective until **logMsg()** has been installed. After that time, **logMsg()** will be used regardless of whether **pciLogMsgFunc** has been specified. This allows a polled-mode **kprintf()** to be used before the serial line has been initialized, and **logMsg()** to be used afterward.

The following code from **sysBusPci.c** demonstrates one way to handle PCI configuration messages. This code will save the output messages to a RAM buffer, and allow the messages to be printed after the system has finished booting.

```
/* variable and structure declarations for saved messages */

#ifdef SAVE_MESSAGES

#define PCI_MSG_MAX_SIZE      100
#define PCI_MSG_NUM_MSGS 250

typedef struct pciMsgBuf
    {
    char data[PCI_MSG_MAX_SIZE];
    } PCI_MSG_BUF;

PCI_MSG_BUF     pMsgMem[PCI_MSG_NUM_MSGS];
int             msgIndx = 0;
#endif /* SAVE_MESSAGES */

/* subroutines */

#ifdef SAVE_MESSAGES
/*
 * These are the routines which actually save the messages to
 * memory and print the contents of the messages after the
 * system has booted.
 */

/******************************************************************************
 *
 * pciMsgLogFunc - save PCI diagnostic messages for later perusal
 *
 * This function accepts arguments in the same format as logMsg(), and
 * saves them to a RAM buffer.
 *
 *  RETURNS: the number of characters saved to memory.
 */

int pciMsgLogFunc(char *fmt, int a1, int a2, int a3, int a4,
               int a5, int a6)
    {
    char *pData;

    if ( msgIndx >= PCI_MSG_NUM_MSGS )
        return(0);

    pData = (char *)&pMsgMem[msgIndx++].data[0];

    /*
     * we should check for buffer overrun here, but it isn't
     * likely that a buffer overrun will affect anything
     * except the output of pciMsgLogDump().  If an overrun
     * occurs on output line #50, then when pciMsgLogDump()
     * tries to print line #50, it will actually end up
     * printing the first PCI_MSG_MAX_LEN characters of line
     * #50, followed by the entire output of line #51.  It
     * will then print line #50 on the next iteration.  So the
     * only effect is that part of one line will be truncated,
     * and the next line will be displayed twice.  If we were
     * using a linked list instead of an array, this would not
     * be the case, however that would require the memory
     * allocation library to be initialized, which may not be
     * the case.
     */
    return(sprintf(pData, fmt, a1,a2,a3,a4,a5,a6));
    }

/******************************************************************************
 *
```

```
 * pciMsgLogDump - display previously-saved PCI diagnostic messages.
 *
 * This function displays messages saved by pciMsgLogFunc().
 *
 * ARGUMENTS:
 *     beginLoc: the index of the first location to display.
 *     endLoc: the index of the first location which will not be
 *          displayed.  If endLoc is zero, all remaining messages
 *          will be displayed, so pciMsgLogDump(0,0) will display
 *          all saved messages.
 *
 *  RETURNS: N/A
 */

void pciMsgLogDump
    (
    int beginLoc,
    int endLoc
    )
    {
    int i;
    char *pData;
    if ( endLoc == 0 )
        endLoc = msgIndx;

    for ( i = beginLoc ; i < endLoc ; i++ )
        {
        pData = (char *)&pMsgMem[i].data[0];
        printf("[pci %03d] %s", i, pData);
        }
    }
#endif /* SAVE_MESSAGES */
```

Later, where the PCI autoconfiguration functions are actually called from inside **sysPciAutoConfig()**, the calls should look something like the following:

```
/*
 * in sysBusPci.c, the function sysPciAutoConfig() will make
 * the calls to pciAutoConfigLib functions.  This starts with
 * pciAutoConfigLibInit().
 */

/* initialize pciAutoConfigLib */
pPciCookie = pciAutoConfigLibInit(NULL);

#ifdef SAVE_MESSAGES
    /* configure message output */
    pciAutoCfgCtl(pPciCookie, PCI_MSG_LOG_SET, pciMsgLogFunc);
#endif /* SAVE_MESSAGES */
```

The code above, from **sysBusPci.c** will save the messages, but it will not display the output. The output can be displayed manually when the system has completed booting by calling the function **pciMsgLogDump(0,0)**. Or, the system can be configured to display the PCI messages automatically by adding a call to **pciMsgLogDump(0,0)** from an appropriate place in the system startup code.

D functionPreConfigRtn

Like **bridgePreConfigInit()**, the new **functionPreConfigRtn()** can be used to support non-standard devices with a custom callback routine called before configuring the specific device.

Use of devices such as this should be discouraged. However, it may be useful to allow software development during hardware design and debug. To allow these devices to be used, pciAutoConfigLib includes a hook which will be run during the second pass, so that the bus numbers have all been set, but the specific device has not yet been configured. The **functionPreConfigRtn()** hook can be written and installed so that it will perform the actions required by such non-standard devices.

```
/******************************************************************************
*
* pciFunctionPreCfgRtn - custom pre-configuration code for each PCI function
*
* This routine performs any custom configuration required.  It is called
* once for each PCI function discovered on the bus.  It is the responsibility
* of this function to determine whether or not the specified function requires
* custom initialization.
*
* RETURNS: N/A.
*/

void pciFunctionPreCfgRtn
    (
    PCI_SYSTEM *      pSystem,
    UINT              bus,
    UINT              device,
    UINT              function
    )
    {
    return;
    }
```

PTE Minimization

As specified above, PCI memory and I/O space which is configured into the system may require the MMU to map that region, and thus use system memory to hold Page Table Entries (PTEs). This can require significant resources.

In general, consumption of these resources should be considered appropriate use. The BSP developer should have a good understanding of the resources necessary for the system to function, and should not allocate much more than the required space plus the space required for alignment. However, there may be situations where the system design requires a large amount of PCI memory available, where the memory will be unused most of the time. In this case, the memory used for PTEs may be considered excessive. When this happens, the BSP developer or project manager may elect to minimize the resources consumed by PTEs by manipulating the amount of memory controlled by the MMU.

If the processor architecture allows for some areas to be available but unmanaged by the MMU, this strategy can be adopted. For example, the PowerPC architecture allows Base Address Translation registers (BATs) to be programmed so that specific memory regions are available but not managed by the MMU. This will minimize the resources required for PTEs.

When all other options have been shown to be inadequate, it may be necessary for the BSP developer to adjust the system tables so as to minimize the overhead of PTEs by just mapping the resources which are actually used, rather than the resources which are made available. Because this requires linkage between two unrelated modules, the PCI configuration system and the memory configuration system, it should be avoided if at all possible.

To accomplish this linkage, call **pciAutoCfg()** or **pciAutoConfig()** before the MMU is initialized, allow large amounts of PCI resources to be made available, and then adjust the system tables so as to minimize the overhead of PTEs by just mapping the resources which are actually used. The BSP developer will need to know the index of the sysPhysMemDesc[] table entries for each of the PCI resources. Care must be taken so that subsequent careless modification of sysPhysMemDesc[] do not cause failures in unrelated areas of the system.

Here is a description of how to reduce the sysPhysMemDesc[] table. First, call **pciAutoCfg()**, as usual, setting options and configuration parameters as appropriate. This must be done from within **sysHwInit()**, before the MMU is initialized. After **pciAutoCfg()** is finished, query each resource type in turn, using **pciAutoCfgCtl()** with the macros **PCI_MEM32_SIZE_GET**, **PCI_MEMIO32_SIZE_GET**, **PCI_IO32_SIZE_GET**, and **PCI_IO16_SIZE_GET**. From these figures, you need to adjust the sysPhysMemDesc[] entry. To make this easier, you may wish to use macros within the definition of sysPhysMemDesc[], for example:

```
PHYS_MEM_DESC sysPhysMemDesc [] =
    {
#define SYS_PHYS_MEM_INDEX_VEC_TBL   0

    {
    /* Vector Table and Interrupt Stack */
    ...
    },


#define SYS_PHYS_MEM_INDEX_LOCAL_DRAM       (SYS_PHYS_MEM_INDEX_VEC_TBL+1)
    /* Local DRAM - Must be second entry in sysPhysMemDesc for Auto Sizing */
    {
    ...
    },

#define SYS_PHYS_MEM_INDEX_PCI_ISA          (SYS_PHYS_MEM_INDEX_LOCAL_DRAM+1)
    /* Access to PCI ISA I/O space */
    {
    ...
    },

#define SYS_PHYS_MEM_INDEX_PCI_IO32          (SYS_PHYS_MEM_INDEX_PCI_ISA+1)
    /* Access to PCI I/O space */
    {
    ...
    },

#define SYS_PHYS_MEM_INDEX_PCI_MEMIO32      (SYS_PHYS_MEM_INDEX_PCI_IO32+1)
    /* Access to PCI ISA memory space */
    {
    ...
    },

#define SYS_PHYS_MEM_INDEX_PCI_MEM32        (SYS_PHYS_MEM_INDEX_PCI_MEMIO32+1)
    /* Access to PCI memory space */
    {
    ...
    },

    ...

    };
```

With these definitions, the PCI configuration code following the **pciAutoCfg()** call would adjust the sysPhysMemDesc[] fields. For example, the code to adjust the prefetchable memory space might look like this:

```
    /* minimize PTE requirements for PCI memory spaces */

    /* find amount of memio32 space used */
```

```
pciAutoCfgCtl(pPciCookie, PCI_MEMIO32_SIZE_GET, &memIo32Used);
/* align to page size */
memIo32Used |= (VM_PAGE_SIZE - 1); /* VM_PAGE_SIZE must be power of 2 */
memIo32Used++;
if ( memIo32Used <= VM_PAGE_SIZE )
   {
   memIo32Used = VM_PAGE_SIZE;
   pciMsgLogFunc("memIo: Specifying 1 page\n",
                  1,2,3,4,5,6);
   }
else if ( memIo32Used > PCI_MSTR_MEMIO_SIZE )
   {
   pciMsgLogFunc("mem: was 0x%x now 0x%x -- ERROR: Over Allotment!\n",
               sysPhysMemDesc[SYS_PHYS_MEM_INDEX_PCI_MEM].len,
               mem32Used,
               3,4,5,6);
   }
else
   {
   pciMsgLogFunc("memIo: was 0x%x now 0x%x\n",
               sysPhysMemDesc[SYS_PHYS_MEM_INDEX_PCI_MEMIO].len,
               memIo32Used,
               3,4,5,6);
   }
sysPhysMemDesc[SYS_PHYS_MEM_INDEX_PCI_MEMIO].len = memIo32Used;
```

With the above, large amounts of PCI resources can be made available with minimal impact on system resources caused by PTE usage for the PCI address ranges which are not actually allocated.

The above examples are specified for the new API. The same can be accomplished using the older **pciAutoConfig()** interface. When using this interface, the **PCI_SYSTEM** structure is updated, and the values of io16Used, io32Used, memIo32Used, and mem32Used can be calculated as follows, exemplified by mem32Used:

```
mem32Used = PCI_MSTR_MEM_SIZE - sysParams.pciMem32Size;
mem32Used |= 0x00000fff;
mem32Used++;
```

NOTE: The strategy to modify **sysPhysMemDesc[]** to reduce resource usage is difficult to maintain, due to the inter-relationship between **sysPhysMemDesc[]** and the PCI configuration software, and is not recomended.

Miscellany

The function **pciAutoCfg()** can be called multiple times, for example, once during initialization of bootrom and a second time during initialization of the final vxWorks image. There should be no problems running **pciAutoCfg()** multiple times, provided that it is not called during normal system operation. Specifically, it can safely be called once by the boot system startup and a second time for normal vxWorks operation.

However, some BSPs have a check to make sure that PCI configuration is not performed multiple times. There are some times when this can cause problems, such as when debugging PCI configurations or whenever there is the possibility of PCI configuration changes between product releases. For these situations, it would be best to disable the check and force PCI configuration to occur when the OS boots, regardless of whether PCI configuration has been performed earlier.

嵌入式资源免费下载

总线协议：

1. 基于 PCIe 驱动程序的数据传输卡 DMA 传输
2. 基于 PCIe 总线协议的设备驱动开发
3. CANopen 协议介绍
4. 基于 PXI 总线 RS422 数据通信卡 WDM 驱动程序设计
5. FPGA 实现 PCIe 总线 DMA 设计
6. PCI Express 协议实现与验证
7. VPX 总线技术及其实现
8. 基于 Xilinx FPGA 的 PCIE 接口实现
9. 基于 PCI 总线的 GPS 授时卡设计
10. 基于 CPCI 标准的 6U 信号处理平台的设计
11. USB30 电路保护
12. USB30 协议分析与框架设计
13. USB 30 中的 CRC 校验原理及实现
14. 基于 CPLD 的 UART 设计
15. IPMI 在 VPX 系统中的应用与设计
16. 基于 CPCI 总线的 PMC 载板设计
17. 基于 VPX 总线的工件台运动控制系统研究与开发
18. PCI Express 流控机制的研究与实现
19. UART16C554 的设计
20. 基于 VPX 的高性能计算机设计
21. 基于 CAN 总线技术的嵌入式网关设计
22. Visual C 串行通讯控件使用方法与技巧的研究
23. IEEE1588 精密时钟同步关键技术研究
24. GPS 信号发生器射频模块的一种实现方案
25. 基于 CPCI 接口的视频采集卡的设计
26. 基于 VPX 的 3U 信号处理平台的设计
27. 基于 PCI Express 总线 1394b 网络传输系统 WDM 驱动设计
28. AT89C52 单片机与 ARINC429 航空总线接口设计
29. 基于 CPCI 总线多 DSP 系统的高速主机接口设计
30. 总线协议中的 CRC 及其在 SATA 通信技术中的应用
31. 基于 FPGA 的 SATA 硬盘加解密控制器设计
32. Modbus 协议在串口通讯中的研究及应用
33. 高可用的磁盘阵列 Cache 的设计和实现
34. RAID 阵列中高速 Cache 管理的优化

35. 一种新的基于 RAID 的 CACHE 技术研究与实现
36. 基于 PCIE-104 总线的高速数据接口设计
37. 基于 VPX 标准的 RapidIO 交换和 Flash 存储模块设计
38. 北斗卫星系统在海洋工程中的应用
39. 北斗卫星系统在远洋船舶上应用的研究
40. 基于 CPCI 总线的红外实时信号处理系统
41. 硬件实现 RAID 与软件实现 RAID 的比较
42. 基于 PCI Express 总线系统的热插拔设计
43. 基于 RAID5 的磁盘阵列 Cache 的研究与实现
44. 基于 PCI 总线的 MPEG2 码流播放卡驱动程序开发
45. 基于磁盘异或引擎的 RAID5 小写性能优化
46. 基于 IEEE1588 的时钟同步技术研究
47. 基于 Davinci 平台的 SD 卡读写优化
48. 基于 PCI 总线的图像处理及传输系统的设计
49. 串口和以太网通信技术在油液在线监测系统中的应用
50. USB30 数据传输协议分析及实现
51. IEEE 1588 协议在工业以太网中的实现
52. 基于 USB30 的设备自定义请求实现方法
53. IEEE1588 协议在网络测控系统中的应用
54. USB30 物理层中弹性缓冲的设计与实现
55. USB30 的高速信息传输瓶颈研究
56. 基于 IPv6 的 UDP 通信的实现
57. 一种基于 IPv6 的流媒体传送方案研究与实现
58. 基于 IPv4-IPv6 双栈的 MODBUS-TCP 协议实现
59. RS485CAN 网关设计与实现
60. MVB 周期信息的实时调度
61. RS485 和 PROFINET 网关设计
62. 基于 IPv6 的 Socket 通信的实现
63. MVB 网络重复器的设计
64. 一种新型 MVB 通信板的探究
65. 具有 MVB 接口的输入输出设备的分析
66. 基于 STM32 的 GSM 模块综合应用
67. 基于 ARM7 的 MVB CAN 网关设计
68. 机车车辆的 MVB CAN 总线网关设计
69. 智能变电站冗余网络中 IEEE1588 协议的应用
70. CAN 总线的浅析 CANopen 协议
71. 基于 CANopen 协议实现多电机系统实时控制
72. 以太网时钟同步协议的研究
73. 基于 CANopen 的列车通信网络实现研究
74. 基于 SJA1000 的 CAN 总线智能控制系统设计
75. 基于 CANopen 的运动控制单元的设计
76. 基于 STM32F107VC 的 IEEE 1588 精密时钟同步分析与实现

77. 分布式控制系统精确时钟同步技术
78. 基于 IEEE 1588 的时钟同步技术在分布式系统中应用
79. 基于 SJA1000 的 CAN 总线通讯模块的实现
80. 嵌入式设备的精确时钟同步技术的研究与实现
81. 基于 SJA1000 的 CAN 网桥设计
82. 基于 CAN 总线分布式温室监控系统的设计与实现
83. 基于 DSP 的 CANopen 通讯协议的实现
84. 基于 PCI9656 控制芯片的高速网卡 DMA 设计
85. 基于以太网及串口的数据采集模块设计

VxWorks：

1. 基于 VxWorks 的多任务程序设计
2. 基于 VxWorks 的数据采集存储装置设计
3. Flash 文件系统分析及其在 VxWorks 中的实现
4. VxWorks 多任务编程中的异常研究
5. VxWorks 应用技巧两例
6. 一种基于 VxWorks 的飞行仿真实时管理系统
7. 在 VxWorks 系统中使用 TrueType 字库
8. 基于 FreeType 的 VxWorks 中文显示方案
9. 基于 Tilcon 的 VxWorks 简单动画开发
10. 基于 Tilcon 的某武器显控系统界面设计
11. 基于 Tilcon 的综合导航信息处理装置界面设计
12. VxWorks 的内存配置和管理
13. 基于 VxWorks 系统的 PCI 配置与应用
14. 基于 MPC8270 的 VxWorks BSP 的移植
15. Bootrom 功能改进经验谈
16. 基于 VxWorks 嵌入式系统的中文平台研究与实现
17. VxBus 的 A429 接口驱动
18. 基于 VxBus 和 MPC8569E 千兆网驱动开发和实现
19. 一种基于 vxBus 的 PPC 与 FPGA 高速互联的驱动设计方法
20. 基于 VxBus 的设备驱动开发
21. 基于 VxBus 的驱动程序架构分析
22. 基于 VxBus 的高速数据采集卡驱动程序开发
23. Vxworks 下的冗余 CAN 通讯模块设计
24. WindML 工业平台下开发 S1d13506 驱动及显示功能的实现
25. WindML 中 Mesa 的应用
26. VxWorks 下图形用户界面开发中双缓冲技术应用

27. VxWorks 上的一种 GUI 系统的设计与实现
28. VxWorks 环境下 socket 的实现
29. VxWorks 的 WindML 图形界面程序的框架分析
30. VxWorks 实时操作系统及其在 PC104 下以太网编程的应用
31. 实时操作系统任务调度策略的研究与设计
32. 军事指挥系统中 VxWorks 下汉字显示技术
33. 基于 VxWorks 实时控制系统中文交互界面开发平台
34. 基于 VxWorks 操作系统的 WindML 图形操控界面实现方法
35. 基于 GPU FPGA 芯片原型的 VxWorks 下驱动软件开发
36. VxWorks 下的多串口卡设计
37. VxWorks 内存管理机制的研究
38. T9 输入法在 Tilcon 下的实现
39. 基于 VxWorks 的 WindML 图形界面开发方法
40. 基于 Tilcon 的 IO 控制板可视化测试软件的设计和实现
41. 基于 VxWorks 的通信服务器实时多任务软件设计
42. 基于 VXWORKS 的 RS485MVB 网关的设计与实现
43. 实时操作系统 VxWorks 在微机保护中的应用
44. 基于 VxWorks 的多任务程序设计及通信管理
45. 基于 Tilcon 的 VxWorks 图形界面开发技术
46. 嵌入式图形系统 Tilcon 及应用研究
47. 基于 VxWorks 的数据采集与重演软件的图形界面的设计与实现
48. 基于嵌入式的 Tilcon 用户图形界面设计与开发
49. 基于 Tilcon 的交互式多页面的设计
50. 基于 Tilcon 的嵌入式系统人机界面开发技术
51. 基于 Tilcon 的指控系统多任务人机交互软件设计
52. 基于 Tilcon 航海标绘台界面设计
53. 基于 Tornado 和 Tilcon 的嵌入式 GIS 图形编辑软件的开发
54. VxWorks 环境下内存文件系统的应用
55. VxWorks 下的多重定时器设计
56. Freescale 的 MPC8641D 的 VxWorks BSP
57. VxWorks 实验五[时间片轮转调度]
58. 解决 VmWare 下下载大型工程.out 出现 WTX Error 0x100de 的问题
59.

Linux:

1. Linux 程序设计第三版及源代码
2. NAND FLASH 文件系统的设计与实现
3. 多通道串行通信设备的 Linux 驱动程序实现

4. Zsh 开发指南-数组
5. 常用 GDB 命令中文速览
6. 嵌入式 C 进阶之道
7. Linux 串口编程实例
8. 基于 Yocto Project 的嵌入式应用设计
9. Android 应用的反编译
10. 基于 Android 行为的加密应用系统研究
11. 嵌入式 Linux 系统移植步步通
12. 嵌入式 CC++语言精华文章集锦
13. 基于 Linux 的高性能服务器端的设计与研究
14. S3C6410 移植 Android 内核
15. Android 开发指南中文版
16. 图解 Linux 操作系统架构设计与实现原理（第二版）
17. 如何在 Ubuntu 和 Linux Mint 下轻松升级 Linux 内核
18. Android 简单 mp3 播放器源码
19. 嵌入式 Linux 系统实时性的研究
20. Android 嵌入式系统架构及内核浅析
21. 基于嵌入式 Linux 操作系统内核实时性的改进方法研究
22. Linux TCP IP 协议详解
23. Linux 桌面环境下内存去重技术的研究与实现
24. 掌握 Android 7.0 新增特性 Quick Settings
25. Android 应用逆向分析方法研究
26. Android 操作系统的课程教学
27. Android 智能手机操作系统的研究
28. Android 英文朗读功能的实现
29. 基于 Yocto 订制嵌入式 Linux 发行版
30. 基于嵌入式 Linux 的网络设备驱动设计与实现
31. 如何高效学习嵌入式
32. 基于 Android 平台的 GPS 定位系统的设计与实现
33. LINUX ARM 下的 USB 驱动开发
34. Linux 下基于 I2C 协议的 RTC 驱动开发
35. 嵌入式下 Linux 系统设备驱动程序的开发
36. 基于嵌入式 Linux 的 SD 卡驱动程序的设计与实现
37. Linux 系统中进程调度策略
38. 嵌入式 Linux 实时性方法
39. 基于实时 Linux 计算机联锁系统实时性分析与改进
40. 基于嵌入式 Linux 下的 USB30 驱动程序开发方法研究
41. Android 手机应用开发之音乐资源播放器
42. Linux 下以太网的 IPv6 隧道技术的实现
43. Research and design of mobile learning platform based on Android
44. 基于 linux 和 Qt 的串口通信调试器调的设计及应用
45. 在 Linux 平台上基于 QT 的动态图像采集系统的设计

46. 基于 Android 平台的医护查房系统的研究与设计
47. 基于 Android 平台的软件自动化监控工具的设计开发
48. 基于 Android 的视频软硬解码及渲染的对比研究与实现
49. 基于 Android 移动设备的加速度传感器技术研究
50. 基于 Android 系统振动测试仪研究
51. 基于缓存竞争优化的 Linux 进程调度策略
52. Linux 基于 W83697 和 W83977 的 UART 串口驱动开发文档

Windows CE：

1. Windows CE.NET 下 YAFFS 文件系统 NAND Flash 驱动程序设计
2. Windows CE 的 CAN 总线驱动程序设计
3. 基于 Windows CE.NET 的 ADC 驱动程序实现与应用的研究
4. 基于 Windows CE.NET 平台的串行通信实现
5. 基于 Windows CE.NET 下的 GPRS 模块的研究与开发
6. win2k 下 NTFS 分区用 ntldr 加载进 dos 源代码
7. Windows 下的 USB 设备驱动程序开发
8. WinCE 的大容量程控数据传输解决方案设计
9. WinCE6.0 安装开发详解
10. DOS 下仿 Windows 的自带计算器程序 C 源码
11. G726 局域网语音通话程序和源代码
12. WinCE 主板加载第三方驱动程序的方法
13. WinCE 下的注册表编辑程序和源代码
14. WinCE 串口通信源代码
15. WINCE 的 SD 卡程序[可实现读写的源码]
16. 基于 WinCE 的 BootLoader 研究
17. Windows CE 环境下无线网卡的自动安装
18. 基于 Windows CE 的可视电话的研究与实现
19. 基于 WinCE 的嵌入式图像采集系统设计
20. 基于 ARM 与 WinCE 的掌纹鉴别系统
21. DCOM 协议在网络冗余环境下的应用
22. Windows XP Embedded 在变电站通信管理机中的应用
23. XPE 在多功能显控台上的开发与应用
24. 基于 Windows XP Embedded 的 LKJ2000 仿真系统设计与实现
25. 虚拟仪器的 Windows XP Embedded 操作系统开发
26. 基于 EVC 的嵌入式导航电子地图设计
27. 基于 XPEmbedded 的警务区 SMS 指挥平台的设计与实现
28. 基于 XPE 的数字残币兑换工具开发
29. Windows CENET 下 ADC 驱动开发设计

30. Windows CE 下 USB 设备流驱动开发与设计
31. Windows 驱动程序设计
32. 基于 Windows CE 的 GPS 应用
33. 基于 Windows CE 下大像素图像分块显示算法的研究
34. 基于 Windows CE 的数控软件开发与实现

PowerPC：

1. Freescale MPC8536 开发板原理图
2. 基于 MPC8548E 的固件设计
3. 基于 MPC8548E 的嵌入式数据处理系统设计
4. 基于 PowerPC 嵌入式网络通信平台的实现
5. PowerPC 在车辆显控系统中的应用
6. 基于 PowerPC 的单板计算机的设计
7. 用 PowerPC860 实现 FPGA 配置
8. 基于 MPC8247 嵌入式电力交换系统的设计与实现
9. 基于设备树的 MPC8247 嵌入式 Linux 系统开发
10. 基于 MPC8313E 嵌入式系统 UBoot 的移植
11. 基于 PowerPC 处理器 SMP 系统的 UBoot 移植
12. 基于 PowerPC 双核处理器嵌入式系统 UBoot 移植
13. 基于 PowerPC 的雷达通用处理机设计
14. PowerPC 平台引导加载程序的移植
15. 基于 PowerPC 嵌入式内核的多串口通信扩展设计
16. 基于 PowerPC 的多网口系统抗干扰设计
17. 基于 MPC860T 与 VxWorks 的图形界面设计
18. 基于 MPC8260 处理器的 PPMC 系统
19. 基于 PowerPC 的控制器研究与设计
20. 基于 PowerPC 的模拟量输入接口扩展
21. 基于 PowerPC 的车载通信系统设计
22. 基于 PowerPC 的嵌入式系统中通用 IO 口的扩展方法
23. 基于 PowerPC440GP 型微控制器的嵌入式系统设计与研究
24. 基于双 PowerPC 7447A 处理器的嵌入式系统硬件设计
25. 基于 PowerPC603e 通用处理模块的设计与实现
26. 嵌入式微机 MPC555 驻留片内监控器的开发与实现
27. 基于 PowerPC 和 DSP 的电能质量在线监测装置的研制
28. 基于 PowerPC 架构多核处理器嵌入式系统硬件设计
29. 基于 PowerPC 的多屏系统设计
30. 基于 PowerPC 的嵌入式 SMP 系统设计

ARM：

1. 基于 DiskOnChip 2000 的驱动程序设计及应用
2. 基于 ARM 体系的 PC-104 总线设计
3. 基于 ARM 的嵌入式系统中断处理机制研究
4. 设计 ARM 的中断处理
5. 基于 ARM 的数据采集系统并行总线的驱动设计
6. S3C2410 下的 TFT LCD 驱动源码
7. STM32 SD 卡移植 FATFS 文件系统源码
8. STM32 ADC 多通道源码
9. ARM Linux 在 EP7312 上的移植
10. ARM 经典 300 问
11. 基于 S5PV210 的频谱监测设备嵌入式系统设计与实现
12. Uboot 中 start.S 源码的指令级的详尽解析
13. 基于 ARM9 的嵌入式 Zigbee 网关设计与实现
14. 基于 S3C6410 处理器的嵌入式 Linux 系统移植
15. CortexA8 平台的μC-OSⅡ及 LwIP 协议栈的移植与实现
16. 基于 ARM 的嵌入式 Linux 无线网卡设备驱动设计
17. ARM S3C2440 Linux ADC 驱动
18. ARM S3C2440 Linux 触摸屏驱动
19. Linux 和 Cortex-A8 的视频处理及数字微波传输系统设计
20. Nand Flash 启动模式下的 Uboot 移植
21. 基于 ARM 处理器的 UART 设计
22. ARM CortexM3 处理器故障的分析与处理
23. ARM 微处理器启动和调试浅析
24. 基于 ARM 系统下映像文件的执行与中断运行机制的实现
25. 中断调用方式的 ARM 二次开发接口设计
26. ARM11 嵌入式系统 Linux 下 LCD 的驱动设计
27. Uboot 在 S3C2440 上的移植
28. 基于 ARM11 的嵌入式无线视频终端的设计
29. 基于 S3C6410 的 Uboot 分析与移植
30. 基于 ARM 嵌入式系统的高保真无损音乐播放器设计
31. UBoot 在 Mini6410 上的移植
32. 基于 ARM11 的嵌入式 Linux NAND FLASH 模拟 U 盘挂载分析与实现
33. 基于 ARM11 的电源完整性分析
34. 基于 ARM S3C6410 的 uboot 分析与移植
35. 基于 S5PC100 移动视频监控终端的设计与实现

Hardware:

1. DSP 电源的典型设计
2. 高频脉冲电源设计
3. 电源的综合保护设计
4. 任意波形电源的设计
5. 高速 PCB 信号完整性分析及应用
6. DM642 高速图像采集系统的电磁干扰设计
7. 使用 COMExpress Nano 工控板实现 IP 调度设备
8. 基于 COM Express 架构的数据记录仪的设计与实现
9. 基于 COM Express 的信号系统逻辑运算单元设计
10. 基于 COM Express 的回波预处理模块设计
11. 基于 X86 平台的简单多任务内核的分析与实现
12. 基于 UEFI Shell 的 PreOS Application 的开发与研究
13. 基于 UEFI 固件的恶意代码防范技术研究
14. MIPS 架构计算机平台的支持固件研究
15. 基于 UEFI 固件的攻击验证技术研究
16. 基于 UEFI 的 Application 和 Driver 的分析与开发
17. 基于 UEFI 的可信 BIOS 研究与实现
18. 基于 UEFI 的国产计算机平台 BIOS 研究
19. 基于 UEFI 的安全模块设计分析
20. 基于 FPGA Nios II 的等精度频率计设计
21. 基于 FPGA 的 SOPC 设计
22. 基于 SOPC 基本信号产生器的设计与实现
23. 基于龙芯平台的 PMON 研究与开发
24. 基于 X86 平台的嵌入式 BIOS 可配置设计
25. 基于龙芯 2F 架构的 PMON 分析与优化
26. CPU 与 GPU 之间接口电路的设计与实现
27. 基于龙芯 1A 平台的 PMON 源码编译和启动分析
28. 基于 PC104 工控机的嵌入式直流监控装置的设计
29. GPGPU 技术研究与发展
30. GPU 实现的高速 FIR 数字滤波算法
31. 一种基于 CPUGPU 异构计算的混合编程模型
32. 面向 OpenCL 模型的 GPU 性能优化
33. 基于 GPU 的 FDTD 算法
34. 基于 GPU 的瑕疵检测
35. 基于 GPU 通用计算的分析与研究
36. 面向 OpenCL 架构的 GPGPU 量化性能模型
37. 基于 OpenCL 的图像积分图算法优化研究

38. 基于 OpenCL 的均值平移算法在多个众核平台的性能优化研究
39. 基于 OpenCL 的异构系统并行编程
40. 嵌入式系统中热备份双机切换技术研究

Programming:

1. 计算机软件基础数据结构 - 算法
2. 高级数据结构对算法的优化
3. 零基础学算法
4. Linux 环境下基于 TCP 的 Socket 编程浅析
5. Linux 环境下基于 UDP 的 socket 编程浅析
6. 基于 Socket 的网络编程技术及其实现
7. 数据结构考题 - 第 1 章 绪论
8. 数据结构考题 - 第 2 章 线性表
9. 数据结构考题 - 第 2 章 线性表 - 答案
10. 基于小波变换与偏微分方程的图像分解及边缘检测
11. 基于图像能量的布匹瑕疵检测方法
12. 基于 OpenCL 的拉普拉斯图像增强算法优化研究
13. 异构平台上基于 OpenCL 的 FFT 实现与优化
14. 数据结构考题 - 第 4 章 串
15. 数据结构考题 - 第 4 章 串答案
16.

FPGA / CPLD:

1. 一种基于并行处理器的快速车道线检测系统及 FPGA 实现
2. 基于 FPGA 和 DSP 的 DBF 实现
3. 高速浮点运算单元的 FPGA 实现
4. DLMS 算法的脉动阵结构设计及 FPGA 实现
5. 一种基于 FPGA 的 3DES 加密算法实现
6. 可编程 FIR 滤波器的 FPGA 实现
7. 基于 FPGA 的 AES 加密算法的高速实现
8. 基于 FPGA 的精确时钟同步方法
9. 应用分布式算法在 FPGA 平台实现 FIR 低通滤波器
10. 流水线技术在用 FPGA 实现高速 DSP 运算中的应用
11. 基于 FPGA 的 CAN 总线通信节点设计

12. 基于 FPGA 的高速时钟数据恢复电路的实现
13. 基于 FPGA 的高阶高速 FIR 滤波器设计与实现
14. 基于 FPGA 高效实现 FIR 滤波器的研究
15.