Technical Note

Using the DDF Available with ZTP v1.4.x

TN003801-0105

Introduction

This Technical Note discusses the usage of the device driver framework (DDF) available with the ZiLOG TCP/IP Software Suite (ZTP v1.4.x). ZTP is a highly modular and scalable software system which is broadly classified into the following four categories:

- RZK Kernel
- TCP/IP Stack
- Application Protocols
- Demo Programs

The ZTP v1.4.x release comprises of a set of drivers for on-chip peripherals such as Ethernet, UART, I^2C, SPI, and RTC, all of which are a part of ZiLOG's eZ80Acclaim!™ Development Board. Additionally, protocol drivers such as PPP and HDLC also form a part of the ZTP v1.4.x package.

ZTP v1.4.x uses the Real-Time ZiLOG Kernel (RZK), and the ZTP core contains TCP/IP protocols. ZTP supports three connections: the Stream connection (TCP), the Datagram connection (UDP), and the Secure connection (SSL).

A number of ZTP applications are high-level application protocol components that make use of core components to perform standard network functions. An optional run-time Operating System (OS) debugging-shell interface is provided with ZTP, and users can initiate shell commands either locally using an RS-232 terminal or remotely via Telnet. Another component of the ZTP application sub-system is an embedded Flash file system that supports multiple disk volumes, and works with the available Flash and RAM memory. For a detailed information about ZTP v1.4.x and RZK v1.1.x, refer to the related documents listed in the References section on page 9.

Using Device Drivers

This section lists the project settings required to use the device drivers available as a part of the ZTP Board Support Packages (BSPs). The process of adding these device drivers to the device driver table of a system, initializing device drivers, calling the open and close device driver APIs, and reading from and writing to devices is also discussed in detail.

Project Settings Required for Using Device Drivers

To use the device driver APIs that are a part of the ZTP BSP, ensure that the `XTL = 1` definition is added to the **Preprocessor Definitions** category under the **C** tab of the **Project Settings** dialog box. When the project is linked, the code from the device driver libraries is included in the project. If you do not want to use these device driver APIs, then add the `XTL = 0` definition to the **Preprocessor Definitions** category.

Adding Device Drivers to the Device Driver Table of a System

A structure describing the device driver must be added to the system's device driver table for every new device.

In this Technical Note, the following declaration statement located in the `..\ZTP\src\XTL\XTLZdevice.c` file, represents a device driver table.

```
RZK_DEVICE_CB_t usrDevBlk [XTL_MAX_DCBH];
```

Where,

- `userDevBlk` is an array of structure of the type `RZK_DEVICE_CB_t`
- `XTL_MAX_DCBH` is a macro defined in the `..\ZTP\src\XTL\emulator.h` file that specifies the maximum number of device drivers supported

The `XTL = 1` definition listed in the **Preprocessor Definition** field under the **C** tab of the **Project Settings** dialog box, enables users to utilize the default device drivers supported by the system, in addition to other device drivers added to the device driver table of the system.

A prototype of a device driver structure declared in the `..ZTP1.4.0\RZK\eZ80F91\Inc\Core\Zdevice.h` file is listed below.

```
typedef struct RZK_DEVICE_CB_t
{
  char InUse;
  RZK_DEV_NAME_t devName [MAX_DEV_NAME_LENGTH + 1];// Device name

  // Function handlers installed by the driver

  DDF_STATUS_t (*fnInit) (struct RZK_DEVICE_CB_t *);
                                                  // Device control block
  DDF_STATUS_t (*fnStop) (struct RZK_DEVICE_CB_t *);
                                                  // Device control block
  DDF_STATUS_t (*fnOpen) (struct RZK_DEVICE_CB_t *,
                          RZK_DEV_NAME_t *, char *);
  DDF_STATUS_t (*fnClose) (struct RZK_DEVICE_CB_t *);
  DDF_STATUS_t (*fnRead) (struct RZK_DEVICE_CB_t *,
                          RZK_DEV_BUFFER_t *, RZK_DEV_BYTES_t);
```

```
  DDF_STATUS_t (*fnWrite) (struct RZK_DEVICE_CB_t *,
                           RZK_DEV_BUFFER_t *, RZK_DEV_BYTES_t);
  DDF_STATUS_t (*fnGetc)(struct RZK_DEVICE_CB_t *);
  DDF_STATUS_t (*fnPutc)(struct RZK_DEVICE_CB_t *,
                         RZK_DEV_BUFFER_t);
  DDF_STATUS_t (*fnIoctl)(struct RZK_DEVICE_CB_t *,
                           RZK_DEV_BYTES_t, char *, char *);
  RZK_DEV_BYTES_t dvintvector;
  char *dvinputoutput;
  RZK_DEV_MODE_t devMode;     // Driver provides the default
                                 // DevMode.
                                 // RX_INTR/RX_POLL, TX_INTR/TX_POLL
  UINT16 dvminor;
} RZK_DEVICE_CB_t;
```

Depending on the device driver required for a particular application, an appropriate structure similar to the one provided above can be declared by the application developer.

In this Technical Note, the following two code snippets illustrate the creation of a device driver table for an I^2C device, followed by the addition of the I^2C device driver to a global device driver table. For a detailed information about the device driver table, refer to the ZiLOG Real-Time Kernel Reference Manual (RM0006) available on zilog.com.

```
KE_DEV   I2c_driver
{
  0,
  "I2C",
  (FNPTR_RZKDEV_INIT)I2C_Init,
  (FNPTR_RZKDEV_STOP)IONULL,
  (FNPTR_RZKDEV_OPEN)I2C_Open,
  (FNPTR_RZKDEV_CLOSE)I2C_Close,
  (FNPTR_RZKDEV_READ)I2C_Read,
  (FNPTR_RZKDEV_WRITE)I2C_Write,
  (FNPTR_RZKDEV_GETC)IONULL,
  (FNPTR_RZKDEV_PUTC)IONULL,
  (FNPTR_RZKDEV_IOCTL)I2C_Control,
  (RZK_DEV_BYTES_t)IONULL,
  (CHAR*)&i2cCtrl,
  0,
  0
}


// To add the above device driver to the global device driver table.
void function1 (void)
```

```
{
 DID I2cDeviceID;
 MyDeviceID = adddevice (& I2c_driver, 0x1234);
 If (I2cDeviceID == SYSERR)
 {
  Kprintf ("Unable to add my device"\n);
 }
 else
 {
  kprintf("my device ID is %d\n", I2cDeviceID);
 }
}
```

A list of the active device drivers in a system is obtained using the `devs` shell command. This command displays the various device drivers available in the device driver table of the system.

Initializing Device Drivers

After adding structures related to a device driver to the device driver table successfully, it is essential to invoke the initialize routine before any of the device driver services are accessed.

After a new device driver is added to the device driver table and initialized using the `adddevice()` API, any process knowing the device ID of the driver can use the new device driver. Refer to the Calling the Open Device Driver API section on page 5 for details about obtaining a device ID.

The following code snippet illustrates the initialization of drivers for an I^2C device.

```
extern KE_DEV I2c_driver
void function1 (void)
{
 DID MyDeviceID;
 SYSCALL Status;
 MyDeviceID = addevice (& I2cdriver, 0x1234);
 if (MyDeviceID == SYSERR)
 {
  Kprintf ("Unable to add my device\n");
 }
 else
 {
  Status = initialize (MyDeviceID);
 }
 If (Status == OK)
 {
```

```
    Kprintf ("Device initialization successful\n");
  }
}
```

Calling the Open Device Driver API

The `open()` device driver API opens the driver with the specified device ID. The `open()` function is the first function to be invoked in a process requiring the services of the device driver. The `open` function prototype is as shown below.

```
SYSCALL open (DID descrp, char *name, char *mode);
```

The `descrp` parameter in the `open()` function is the value returned by the `adddevice()` function. Whenever a process calls the `open` API, the `FNPTR_RZKDEV_OPEN` routine located in the underlying driver is invoked. The Operating System converts the device ID passed in the `descrp` argument to a pointer that points to the `devsw` structure of the driver. The OS then passes this pointer as a parameter to the `FNPTR_RZKDEV_OPEN` routine of the device driver. The `name` and `mode` parameters are also passed to the `FNPTR_RZKDEV_OPEN` routine, but are not interpreted by the OS.

Within the `FNPTR_RZKDEV_OPEN` routine, the device driver initializes the software resources required to manage a device. The operations performed within the `open()` function are determined by the writer of the device driver. These operations include one or more of the following:

- Clearing buffers
- Initializing queues
- Activating slave device driver
- Allocating required kernel resources

The value returned from the `FNPTR_RZKDEV_OPEN` routine is a valid device ID that can be used as a parameter for other device driver API calls.

The following code snippet illustrates the opening of an I^2C device:

```
extern DID MyDeviceID;
void function2 (void)
{
  DID SlaveDeviceID;
  SlaveDeviceID = open (MyDeviceID, NULLPTR, NULLPTR);
  if (SlaveDeviceID != SYSERR )
  {
    Kprintf ("Ready to transfer data\n");
  }
}
```

Reading from Devices

The `read` routine reads a block of data from the driver with the specified device ID. The buffer is passed to the driver using the `buff` parameter which holds a maximum of `count` data bytes.

The function prototype for reading from a device is as shown below.

```
SYSCALL read (DID descrp, char *buff, WORD count);
```

Whenever a process calls the `read` API, the `FNPTR_RZKDEV_READ` routine in the underlying driver is invoked. The Operating System converts the `descrp` parameter to a pointer that points to the `devsw` structure of the driver, and passes this pointer as a parameter to the `FNPTR_RZKDEV_READ` routine along with the `buff` pointer and the `count` parameter.

Drivers can either copy the currently available data and return control to the caller immediately, or invoke any of the process manipulation or IPC mechanisms of the OS. Drivers can block the caller until the least `count` bytes of data are available.

The following code snippet illustrates the read operation for an I^2C device.

```
BYTE keep_reading_data = TRUE;
PROCESS ReadRoutine( DID MyDeviceID )
{
 INT16 Status;
 char * pBuffer;
 pBuffer = getmem (1000 );
 if( pBuffer == (char*) SYSERR )
 {
  kprintf( "Unable to allocate read buffer\n" );
  return(SYSERR);
 }
 // Call MyDevice to get Rx data. MyDevice blocks this process
 // until data is available.

 while( keep_reading_data == TRUE )
 {
  // Get up to 1000 bytes of data
  Status = read (MyDeviceID, pBuffer, 1000);
  if ( Status > 0 )
  {
   // Process the data.
  }
  if ( Status == SYSERR )
  {
   kprintf( "Read error\n" );
   freemem( pBuffer, 1000 );
   return(SYSERR);
```

```
    }
  }
  freemem( pBuffer, 1000 );
  return(OK);
}
```

Writing to Devices

The `write` routine writes a block of data through the driver with the specified device ID. The buffer is passed to the driver using the `buff` parameter which holds a maximum of `count` data bytes.

The function prototype for writing to a device is as shown below.

```
SYSCALL write (DID descrp, char *buff, WORD count);
```

Whenever a process calls the `write` API, the `FNPTR_RZKDEV_WRITE` routine in the underlying driver is invoked. The OS converts the `descrp` parameter to a pointer that points to the `devsw` structure of the driver, and passes this pointer as a parameter to the `FNPTR_RZKDEV_WRITE` routine together with the `buff` pointer and the `count` parameter.

The following code snippet illustrates the write operation for an I^2C device.

```
void WriteRoutine( DID MyDeviceID, char * pBuffer, WORD Length )
{
  SYSCALL Status;BYTE Done = FALSE;
  pBuffer = getmem( 1000 );
  if( pBuffer == (char*) SYSERR )
  {
    kprintf( "Unable to allocate read buffer\n" );
    return(SYSERR);
  }
  while( !Done )
  {
    Status = write( MyDeviceID, pBuffer, Length );
    if( Status < 0 )
    {
      kprintf( "Error on write %d\n", Status );
      Done = TRUE;
    }
    else{ if( Status < Length )
    {  // Driver is not able to process the entire* buffer.* Wait
       // 100ms before resubmitting remaining* data.*/
      sleep10( 1 );
      pBuffer += Status;
      Length -= Status;
    }
    else
```

```
    {
      Done = TRUE;
    }
  }
}
```

Calling the Close Device Driver API

The `close` routine closes the driver with the specified device ID. Whenever a process calls the `close` API, the `FNPTR_RZKDEV_CLOSE` routine in the underlying driver is invoked.

The Operating System converts the `descrp` parameter to a pointer that points to the `devsw` structure of the driver. The OS then passes this pointer as a parameter to the `FNPTR_RZKDEV_CLOSE` routine.

Within the `FNPTR_RZKDEV_CLOSE` routine, all of the system resources that were acquired during the `FNPTR_RZKDEV_OPEN` function call are released. Drivers that manipulate hardware devices must ensure that the hardware device does not generate interrupts.

The following code snippet illustrates the closing of an I^2C device.

```
extern DID MyDeviceID;
void SetupRoutine (void)
{
  DID SlaveDeviceID, Status;
  SlaveDeviceID = open (MyDeviceID, NULLPTR, NULLPTR);
  if ( SlaveDeviceID != SYSERR )
  {
    kprintf ("Ready to transfer data\n");
    Status = close( SlaveDeviceID);
    if( Status == OK )
    {
      kprintf( "Slave device closed\n" );
    }
  }
}
```

Summary

ZTP includes drivers for devices such as Ethernet, UART, I^2C, SPI, and RTC, all of which are on-chip peripherals of the eZ80Acclaim!™ microcontroller. Although these drivers are specific to the on-chip peripherals of the eZ80Acclaim!™ microcontroller and to devices located on ZiLOG development boards, these drivers can be used as templates to develop software drivers for other devices. The APIs for these device drivers greatly reduce product development time. For details about the device driver framework, please refer to the ZiLOG Real-Time Kernel Reference Manual (RM0006), available on zilog.com.

References

Further details about ZTP v1.4.x, RZK v1.1.x, and the ZiLOG File System can be found in the references listed in Table 1.

Table 1. List of References

Topic	Document Name
RZK v1.1	ZiLOG Real-Time Kernel (RZK) v1.1.0 Product Brief (PB0155)
	ZiLOG Real-Time Kernel (RZK) v1.1.0 Quick Start Guide (QS0048)
	ZiLOG Real-Time Kernel (RZK) v1.1.0 Reference Manual (RM0006)
	ZiLOG Standard Library API Reference Manual (RM0037)
	ZiLOG Real-Time Kernel (RZK) v1.1.0 User Manual (UM0075)
ZTP v1.4	ZiLOG TCP/IP Software Suite (ZTP) v1.4 Quick Start Guide (QS0049)
	ZiLOG TCP/IP Stack v1.4.0 API Reference Manual (RM0040)
	ZiLOG TCP/IP Software Suite (ZTP) v1.4 Reference Manual (RM0041)
ZiLOG File System	ZiLOG File System Quick Start Guide (QS0050)
	ZiLOG File System Reference Manual (RM0039)
	ZiLOG File System User Manual (UM0179)

This publication is subject to replacement by a later edition. To determine whether a later edition exists, or to request copies of publications, contact:

ZiLOG Worldwide Headquarters
532 Race Street
San Jose, CA 95126
Telephone: 408.558.8500
Fax: 408.558.8300
www.zilog.com

Information Integrity

The information contained within this document has been verified according to the general principles of electrical and mechanical engineering. Any applicable source code illustrated in the document was either written by an authorized ZiLOG employee or licensed consultant. Permission to use these codes in any form, besides the intended application, must be approved through a license agreement between both parties. ZiLOG will not be responsible for any code(s) used beyond the intended application. Contact the local ZiLOG Sales Office to obtain necessary license agreements.

Document Disclaimer