TINIS400 External Serial Port Reference Design

This application note describes the implementation of external serial ports on a DSTINIS400 Sockets Board running TINIOS version 1.1x.There are three distinct parts to this implementation: the hardware, RTL, and software.The hardware is implemented as a daughter board, containing a dual UART plugged into a TINI sockets board. Verilog RTL implements the hardware interface from TINI to each of the two UARTs. The software is based on Application Note AN706, Writing a Device Driver for TINIOS, and the existing TINISerialPort class. Implemented as a loadable library, the software includes assembly and Java language code. To simplify the reference design, existing hardware and software APIs are used where possible.

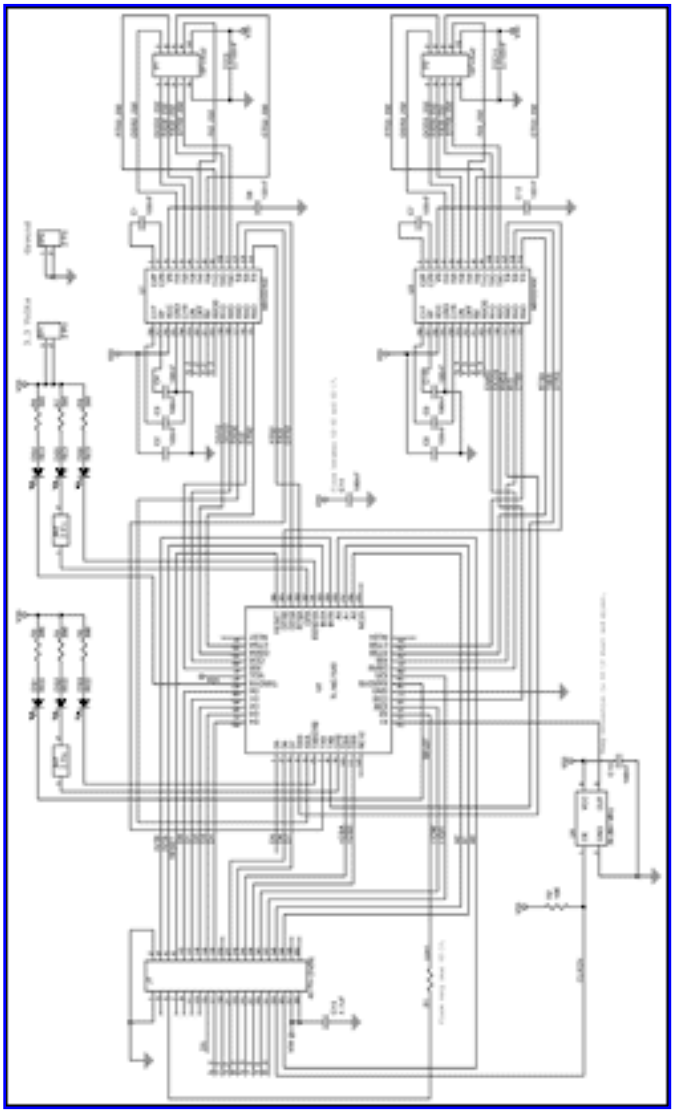
Requirements

The following are requirements of this application note:

1. DSTINIS400 rev. B Sockets Board
    - The Xilinx XC2C64CPLD and associated support hardware must be placed.
2. 87-2C400-SXB hardware referenced in this document, with schematics and layout from the Dallas Semiconductor FTP site.The TL16C752B Dual UART (DUART) from Texas Instruments must operate at 36.864MHz or higher to ensure reliable operation with the DSTINIM400.
3. TINIOS version 1.15 or later
4. Familiarity with hardware design, Verilog, 8051 assembly language, and Java

Hardware

The external serial-port daughter board (Figure 1) is based on a Texas Instruments TL16C752B 3.3V DUART. The address, data, and control lines are routed to the CPLD on the DSTINIs400 through connector J1. To reduce current consumption, the DUART's sleep mode is enabled when not in use. The Maxim MAX3245E transceivers have two status and two control signals, but the status signals are not used in this design. The FORCEON and FORCEOFF signals, routed to the CPLD through J1, power off the device when the port is not in use. A 36.864MHz oscillator is used as the clock input to the DUART. Its OE pin is held in its active state by the CPLD.



(Click for larger size)

Figure 1. DSTINIM400 Serial-Expansion Hardware Schematic (87-2C400-SXB)

RTL

The RTL source is written in Verilog and implemented in a Xilinx XC2C64 CPLD (Figure 2). Xilinx Webpack was used to compile, synthesize, and load the implementation. Several functions, including address decoding, status, and control, are performed using the RTL implementation.

A StatusControl register maps to base address 0a00000h in the TINI Runtime Environment. Aside from CE5, PSEN, and WR, no further decoding is performed. Any read or write from 0a00000h to 0c00000h will access the StatusControl register. Serial ports serial2 and serial3 map to base addresses 600000h and 800000h. The most simple decode logic possible would map the DS80C400's (on the DSTINIM400) address lines A2:A0 to the serial port's address lines S_A2:S_A0. This would accommodate the eight register addresses implemented in each UART. However, this would restrict register access to FIFO-style reads and writes. In other words, if you want to read 32 bytes from the data register, you would have to read 32 times from the same address. These 32 bytes would then typically be written to a sequential block of addresses in the TINI Runtime Environment's heap (the input buffer). In this design, however, A18:A16 are mapped to S_A2:S_A0, giving each register a 64k address range and effectively enabling a read

from anywhere in the 64k range to return the data from the data register regardless of the address.

The UART address-mapping selection was made based on the enhanced data-pointer functionality on the DS80C400. The data pointers have the ability to autotoggle (automatically switch back and forth between two data pointers) and autoincrement (automatically increment the address of the data pointer). You cannot use both the autotoggle and autoincrement features if one data pointer is accessing a FIFO and the other is accessing a sequential buffer. This is because the autotoggle and autoincrement are tied to pairs of data pointers. In most Dallas Semiconductor 8051-based microcontrollers, the code necessary to read from a buffer into another buffer looks like:

```
LoopOrig:
  MOVX A, @DPTR       ; get FIFO byte
  INC DPTR           ; increment dptr
  INC DPS            ; select second dptr
  MOVX @DPTR, A       ; put byte into buffer
  INC DPTR           ; increment pointer
  INC DPS            ; select first dptr
  DJNZ R0, LoopOrig
```

For the DS80C400 and DS80C390 microcontrollers, the following assembly-language instructions take advantage of the the data pointers' autotoggle feature, effectively performing an INC DPS operation as part of the instruction, and thus making the code much more compact, efficient and readable.

```
INC DPTR
MOV DPTR, #data
MOVC A, @A+DPTR
MOVX A, @DPTR
MOVX @DPTR, A
```

This reduces the above code to:

```
LoopAutoToggle:
  MOVX A, @DPTR ; get byte and toggle
  MOVX @DPTR, A ; put byte and toggle
  INC DPTR ; inc pointer and toggle
  INC DPTR ; inc pointer and toggle
  DJNZ R0, LoopAutoToggle
```

Using the DS80c400's autotoggle plus autoincrement features, the code is further reduced to:

```
LoopAutoToggleAutoInc:
```

```
MOVX A, @DPTR ; get, inc and toggle
MOVX @DPTR, A ; put, inc and toggle
DJNZ R0, LoopAutoToggleAutoInc
```

If the hardware is configured in the FIFO access method described above, one of the INC DPTR operations must be removed. In this case, autoincrement is not possible because it works on data pointer pairs. If one of the explicit INC DPTR instructions in the LoopAutoToggle example is removed, then the number of autotoggles is odd, inside the loop, and the roles of the data pointers are reversed after each iteration (i.e., reads, writes, and increments are performed on a different data pointer after each iteration). The only remaining choice for FIFO access is LoopOrig (less the INC DPTR instruction after the FIFO read).

If each UART register maps to span 'n' addresses, then it does not matter if the UART data pointer is incremented, as long as the loops are kept to 'n' iterations or less. This allows the use of the smaller LoopAutoToggleAutoInc code. The UART register address mapping was selected to allow each register to map to a sequential range of addresses on the DS80c400. The 64k size was selected because the CPLD only has access to the DS80C400's {A21:A16, A3:A0} address lines. This was the smallest size that could accommodate the 64 byte FIFO on each UART.

The RTL also performs a control operation and provides status information. Both UART interrupts are tied inside the CPLD to a single interrupt, nExtInt (INT0), on the DS80C400. There are two status bits inside the StatusControl register (StatusControl[1:0]). These bits are tied directly to each of the UART interrupt signals, allowing the driver to identify the source of the interrupt in a single read operation. The control bits (StatusControl[5:2]) are tied to the FORCEON and FORCEOFF inputs on the transceivers. Using these control bits the driver can power up the transceivers when a port is opened, and power them down when a port is closed.

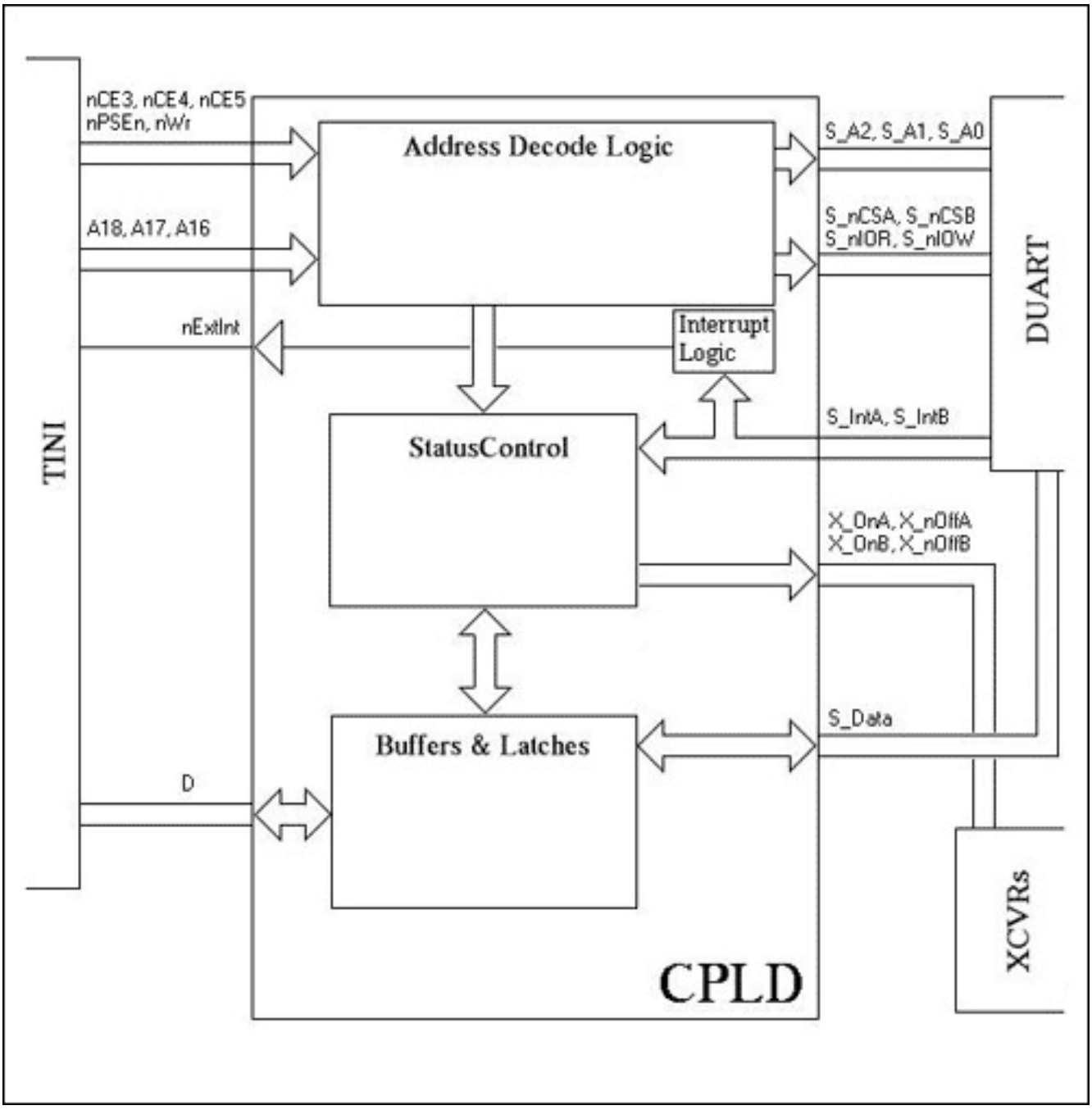


Figure 2. CPLD Interface

Software

The software is designed to run on TINIOS 1.15 or later, and is implemented in two parts: a low-level native driver and higher level Java classes. A loadable TINI native library performs the low-level driver and time-critical operations. Only one instance of this library can be loaded at any one time. Applications requiring multiple external serial ports should be implemented within a single process. The Java class, com.dalsemi.comm.TINIExternalCommDriver, implements the javax.comm interface. This class requires the presence of the /etc/javax.comm.properties file on TINI and must contain the line "Driver=com.dalsemi.comm.TINIExternalCommDriver" to load the library. A more reliable and elegant application would check for the existence of the properties file and generate the file if it does not exist. That would insure the successful loading of the driver, even after the file system is initialized or if the file is erased. The Java class,

com.dalsemi.comm.TINIExternalSerialPort, extends the com.dalsemi.comm.TINISerialPort class (Figure 3). A large portion of this software is based on Application Note AN706, Writing a Device Driver for TINIOS. The details of AN706 are not repeated here, except when additional information provides specific insight to the serial-port driver.

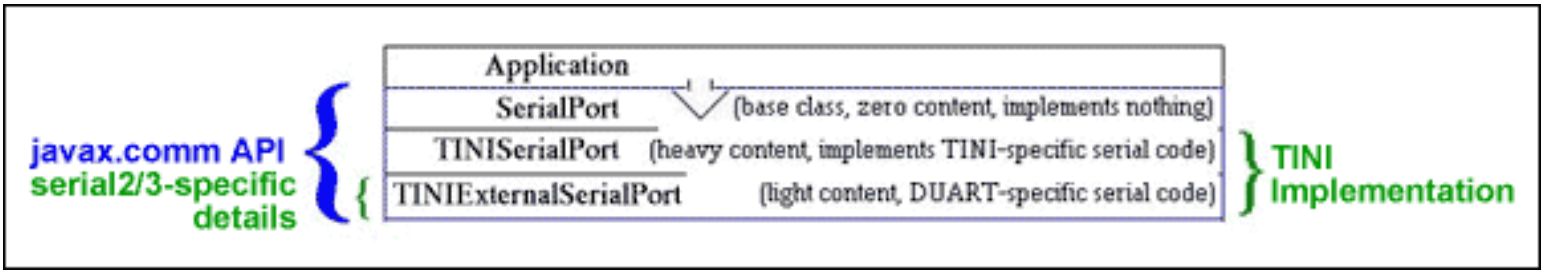


Figure 3. Serial-Port Class Structure

The native and Java libraries for TINIOS 1.15 were expanded to make subclassing the TINISerialPort easier and more compatible with javax.comm wherever possible, and certainly more compatible with other TINIOS serial ports.To implement the low-level functionality associated with the TINIOS.setSerial method, the native fuction System_InstallSetSerialHandler was added. This function takes one parameter, a pointer to a function that installs a handler for the serial port. Passing a #0 (null) to the TINIOS.setSerial method removes the installed handler. The parameters from the Java method TINIOS.setSerial (static int setSerial(int command, int portNum, byte[] data)) are passed to the native handler and returned from the handler unaltered by the OS. For the external serial ports, the functions implemented in the handler are SERIAL_EX_GET_DIVISOR and SERIAL_EX_SET_DIVISOR.

The remaining functions are either implemented in javax.comm or do not make sense in the context of the external serial ports on a DSTINIM400 design. The TINIm390 serial2/3 functions such as SERIAL_EX_SET_PORT_ADDRESS are not required because the driver is loaded as a library and therefore easily changed and recompiled. Also, dynamic variables such as addresses slow code execution. A loadable library with hard-coded addresses offers addresses that can be loaded with a single instruction.

```
MOV DPTR, #SERIAL2_BASE_ADDR
```

The same sequence for a dynamic address would look something like this:

```
MOV DPTR, #SERIAL2_BASE_ADDR
MOVX A, @DPTR
MOV DPL1, A
INC DPTR
MOVX A, @DPTR
MOV DPH1, A
INC DPTR
MOVX A, @DPTR
MOV DPX1, A
```

Most of the library's "Native_" methods provide the same functionality as the original methods implemented in TINISerialPort. Static DB (Define Byte) variables are used by this library for most of the data storage, and reside in the same memory space as the library code. DB access is faster than state block access and does not consume shared resources such as indirects. Larger memory variables like input buffers are malloc-ed from the TINI Runtime Environment's heap, and the pointers to these buffers are stored in static variables. The function Native_TESP_GetVersionNumber was added because the external serial-port driver version is independent of the TINIOS version. Native_GetDriverNumber is necessary because the driver number is required for NativeComm driver calls (see AN706) and native libraries are assigned dynamic driver numbers.

To provide dynamic access to the javax.comm functionality, the TINIExternalCommDriver class is implemented. If javax.comm finds the /etc/javax.comm.properties file, it will attempt to load the driver listed on the "Driver=" line. In this case, the driver is TINIExternalCommDriver. If this class cannot be loaded, then the static TINICommDriver class is loaded. As the external serial port driver is intended to supplement the original serial port driver, not replace it, TINIExternalCommDriver will serve as a proxy for TINICommDriver and load TINISerialPort if serial0, serial1 or serial4 are used (Figure 4).

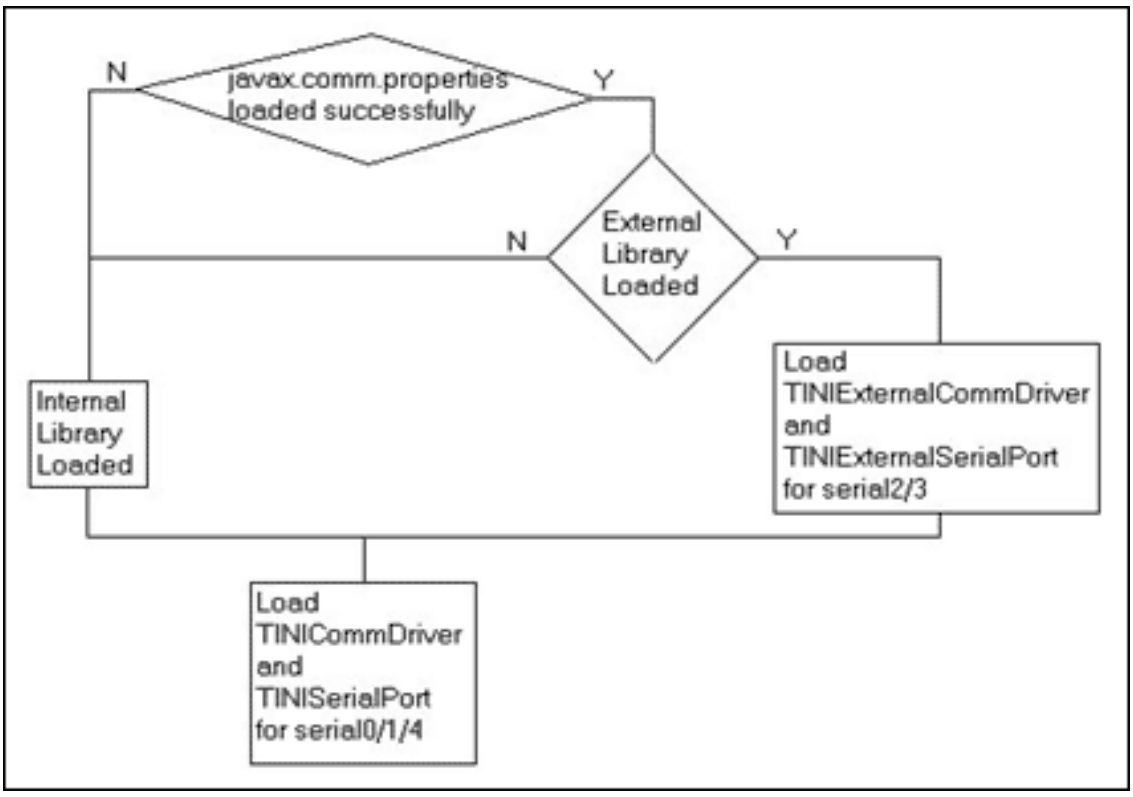


Figure 4. Loading Comm Drivers

The TINIExternalSerialPort class extends the original TINISerialPort class and overloads only a handful of methods. The constructor public TINIExternalSerialPort(String name) has to query the native library for the dynamic driver number. This number is used in a majority of the original methods to make use of the NativeComm driver methods (see AN706).The getPortNames

method was modified to return the new serial-port names "serial2" and "serial3". The TINISerialPort class instantiates an internal class, TININativeThread, to manage an event thread. This class had too many dependencies on the TINIOS core functionality, so a new TINIExternalNativeThread class was added to handle events for serial2/3. The TINIExternalSerialPort methods addEventListener and removeEventListener were written to manage the new event thread. The remaining TINIExternalSerialPort methods are overloaded to handle serial2/3 specific functionality.

Conclusion

Design and implementation of external serial ports on TINI can use of existing hardware and software to reduce the overall development effort. In the reference design presented here, the existing TINIs400 Sockets Board is used along with the optional CPLD to simplify the hardware design by taking advantage of RTL hardware implementation. This approach also provides flexibility for design changes without modifying the existing circuit boards. To handle port-specific issues, the TINISerialPort class is subclassed. Application Note 706, Writing a Device Driver for TINIOS, is used extensively to provide assistance with the native driver development. The application note should be referenced to clarify any issues with this process. By taking advantage of these resources, the bulk of the new effort is confined to designing the hardware and implementing the low-level UART-specific driver functions.

References

Schematics (Source code and binaries–examples/ESerial400 directory of TINI SDK 1.15 or later)
App Note 706: Writing a Device Driver for TINIOS
Microcontroller documentation
DS80C400 data sheet
Java Communications API
User discussion board

More Information

DS80C400:	QuickView	Full (PDF) Data Sheet	Free Samples
DSTINIM400:	QuickView	Full (PDF) Data Sheet	
DSTINIS400:	QuickView	Full (PDF) Data Sheet	
MAX3245E:	QuickView	Full (PDF) Data Sheet	Free Samples