

In this multi-part series we will explore the various ways for 2 or more KronosRobotics chips to communicate. To do this we will be using the various built-in commands using a 2 wire full-duplex connection.

Rather than jumping right in and showing you how to build a multi-chip interface we will take it step by step so that you can understand the whole process.

In Part 1 we will build the slave chip single chip to chip interface. IE 1 master talking to 1 slave. Often we will refer to the slave chip as a Co-Processor or CoProc.

Hardware

We are going to concentrate on the software in this series. You are free to use what ever carrier that works for your application. We used a pair of Athena Workboards populated with Nemesis chip for the development of code.

Step A: Building a Slave Processor

Building and testing the slave processor is very easy and requires very little hookup. In all communications protocols we use the onboard UART built into the various chips. Since we will be using a Nemesis testing is quite easy. The Nemesis program port and UART port are one in the same. This mean we dont need another RS232 connector nor do we need to change any jumpers to test.

The Athena (Nemesis) compiler debug terminal has the ability to send individual 8 bit bytes that enable you to test your Coproc design. We will be sending stadard data packets so use the Send button to transmit the data. The Send Pkt button is used to trans late the data into a special CoProc Protocol that we will get into at a later date. For now we will be using what we call a simple protocol.



In order to see the packet fields you must set the mode to advanced. The mode option is located in the setings menu on the Athena File Manager.

The heart of our simple chip to chip interface is the getpacket command. All Kronos Robotics chips have this command.

getpacket faillabel,indexvarb,valuevarb1,valuevarb2...

To set up the command you must supply various paramaters.

The first parameter is the faillabel. This is where you want the program to go if no packet has been recieved. In other words the all the data in the UART is processed and the packet variables are populated. If all the packet variables have been populated indicating a full packet has been recieved the getpacket command will fall through if not it will jump to the faillabel.

The second parameter a variable that will hold the index to the last packet element read. This variable is how the getpacket command keeps track of its packet progress. The getpacket command is a self syncing command. That is to say that if the getpacket command expects 3 bytes for full packet and it only gets 2 it will automatically reset the index variable back to 0 after a couple milliseconds of idle reception.

The remaining parameters are individual variables used to hold the packet bytes. The number of variables you supply to the command will determine the number of bytes that are expected to fill a packet. For instance if you supply three variable the get packet will expect 3 bytes of packet data to fill these variables. Until all three of those variables have been populated the getpacket command will jump to the faillabel each time it is called.

Lets take a look at a small program that will let us demonstrate the getpacket command.

```
'Chip To Chip 1A1
Nemesis
'Basic IO Proc Demo 1
  dim cmdidx,cmd,data1,data2
  clearall

loop:
  getpacket loop,cmdidx,cmd,data1,data2
  branch cmd,docmd0,docmd1,docmd2
  goto loop

  print "Fall through"
  goto loop

docmd0:
  print "Do Command 0 Data1=",data1," Data2=",data2
  goto loop

docmd1:
  print "Do Command 1 Data1=",data1," Data2=",data2
  goto loop

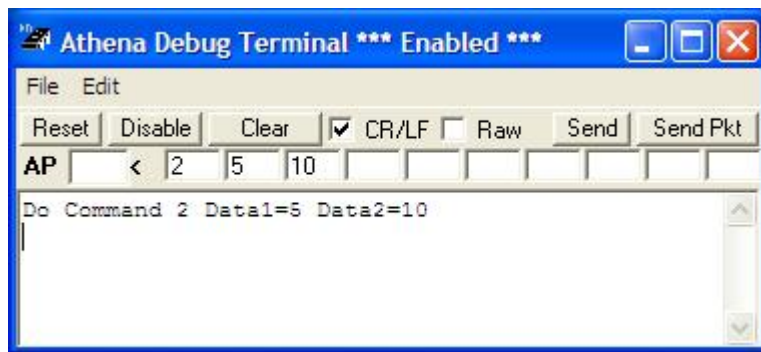
docmd2:
  print "Do Command 2 Data1=",data1," Data2=",data2
  goto loop
```

This program written for the Nemesis will look for a 3 byte packet. Each time getpacket is called it will jump back to loop until the variables cmd,data1, and data 2 have been populated. Most of the time I name the first variable cmd as it is used to tell our program what to do with the remaining packet bytes.

Run the program and send the following three bytes by hitting the send key.



The first byte (2) will branch to the label docmd2 and execute the code found there. Experiment by placing other values into the fields. Notice that if you provide a value to the first byte (cmd) that is greater than the number of jumps provided the branch command will fall through and execute that code.



Now lets create a real Slave processor.

```
'Chip To Chip 1A2
Nemesis
'Basic IO Proc
  dim atodflags,temp,owbyte,x,owport
  dim cmdidx,cmd,data1
  clearall

'-----
'  Main Loop
'-----
loop:
  if RCSTA & 4 <> 0 then
    temp = RCREG 'Clear Frame Error
    cmdidx = 0
  endif
  getpacket loop,cmdidx,cmd,data1
  branch cmd,doinput,dooutput,doread,doreadall,dosetlow,dosethigh,_
  dowriteall,dosetatod,dosetdig,readatod,readatod16
  goto loop

'-----
'  Set Port State (input)
'-----
doinput:
  gosub checkport
  input data1
  goto loop

'-----
'  Set Port State (output)
'-----
dooutput:
  gosub checkport
  output data1
  goto loop

'-----
'  Read port
'-----
doread:
  gosub checkport
  onportgoto data1,-,readhigh
  cptxmt 0
```

```

    goto loop
readhigh:
    cptxmt 1
    goto loop

'-----
' Read all port
'-----

doreadall:
    portget data1
    cptxmt data1
    goto loop

'-----
' Set Port Low
'-----

dosetlow:
    gosub checkport
    low data1
    goto loop

'-----
' Set Port High
'-----

dosethigh:
    gosub checkport
    high data1
    goto loop

'-----
' Set All Ports
'-----

dowriteall:
    portset data1
    goto loop

'-----
' Set Port to AtoD mode
'-----

dosetatod:
    lookup temp,data1,1,2,4,16,32,64,0,8,0,0,0
    atodflags = atodflags ^ temp
    atodinit atodflags,0
    goto loop

'-----
' Set Port to digital
'-----

dosetdig:
    atodinit 0
    goto loop

'-----
' Send AtoD 8-bit
'-----

readatod:
    ATODMODE=0
    atod data1,temp
    cptxmt temp
    goto loop

```

```
'-----  
'Send AtoD 16-bit  
'-----  
readatod16:  
  ATODMODE=128  
  atod data1,temp  
  cptxmt temp  
  temp = ADRESL  
  cptxmt temp  
  goto loop  
  
'-----  
'Utilities  
'-----  
checkport:  
  if data1 > 14 then  
    returnto loop  
  else  
    return  
  endif
```

In this implementation we set up a 2 byte packet. The first byte is the command and the second is used as a data byte. For demonstration purposes we are only accessing the first 8 ports but you could support all the ports if you wish.

- CMD 0,portSet port as input
- CMD 1,portset port as output
- CMD 2,portRead and return port
- CMD 3,0Read and return all ports
- CMD 4,portSet port low
- CMD 5,portSet port high
- CMD 6,dataSet All Ports
- CMD 7,portSet port as Analog
- CMD 8,0Set allports as Digital
- CMD 9,portRead and return Analog value (8 bit)
- CMD 10,portRead and return Analog value (10 bit) Sends 2 bytes

You can set ports 0-7 as input, output or AtoD input. Note that the Nemesis uses port 4 for communications so access to that port is not possible.

Key Program Points

checkport

Notice that we make a call to the subroutine checkport at the beginning of most of the command handlers. This routine just checks to see if the port is out of range. If it is we use the special command returnto to pop the return location off the stack. This allows us to jump back into the loop which is normally a No No from a called subroutine.

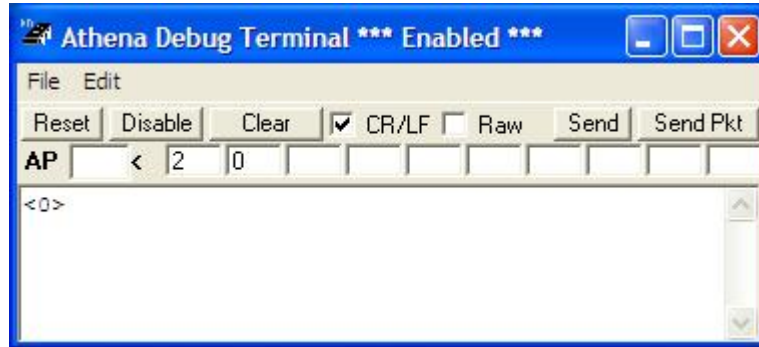
cptxmt

You will see this command used to transmit byte data when requested. You could have used a few other commands as well but the cptxmt command will be used later when we want to support multiprocessors so we might as well use it now.

Error Handler

The if statement between the loop and getpacket command checks for a framing error clears it accordingly.

Using the Debug Terminal send the bytes 2,0. This tells the Slave processor to send the state of port 0. In the example shown below the state is 0. Note that the default state of all the ports is input.



Play around a-bit with the commands. Set a port to AtoD and have it send its reading 8 or 10 bit mode.

Now that the slave processor is built and tested its time to move on to the master processor. We really recommend you test you slave processor before you move on to the master processor.

Step B: Test the Master

First you need to connect the Master chip to the Slave chip. We are going to use ports 0 and 1 as a software serial interface on the Master to connect to the Slave.

Connect:

Master Port 0 to Slave Port 12

Master Port 1 to Slave Port 4

Master Vss to Slave Vss

Note that if you are using chips other than the Nemesis you will need to connect Port 0 on the Master to program RX on the Slave and Port 1 on the Master to Program TX on the Slave.

You wont need your PC connected to the Slave since you have already programmed and tested it. Connect PC to the Master so that you can program it.

Enter the following code into the Master.

```
'Chip To Chip 1B
Nemesis

  dim x
  output 0
  low 0
  pause 25 'Need this to sync Slave
  Setbaud SBAUD9600

loop:
  serout 0,2,0
  serin -,1,x
  debug dec x
  goto loop
```

The program sets up the transmit port 0. The small delay gives the Slave a chance to re-sync after we have set up the port.

We then send the the 2 byte packet 2,0 which tells the Slave to send us the state of port 0. the serin command waits for the value and places it into the x variable.

After displaying the value we go back and do it all again. This pretty much how you communicate with the Slave in a single chip to chip configuration.

In **Part 2** of this series we will show you how to set the Slave up so that you may place more than 1 on a 2 wire bus. Before you proceed you need to understand what we have done here. If not there is little chance you are going to understand **Part 2** as it is much more complicated.

Parts

Athena Microcontroller	Kronos Robotics #16276
Nemesis Microcontroller	Kronos Robotics #16406
Athena WorkBoard Deluxe	Kronos Robotics #16457
Athena Compiler	http://www.kronosrobotics.com/downloads/AthenaSetup.exe