Implementing a DMX512 Universe using the JNIOR Model 410

Bruce S. Cloutier
INTEG Process Group, Inc., bruce.cloutier@integpg.com

Abstract – **The JNIOR Controller is a generic device that is used in a variety of control and monitoring solutions. Application programs are executed by the operating system and are used to optimize the controller for any particular purpose. Some device configurations have achieved widespread use in Digital Cinemas and are now finding application within the Audio Visual, Stage and Lighting markets. A general requirement in some of these implementations involves the control of lighting fixtures. A large segment of fixtures utilize the DMX512 standard for control signals. This paper details how to configure a JNIOR Model 410 to generate a DMX512 Universe capable of controlling multiple DMX lighting fixtures and accessories.**

Index Terms – Lighting Controls, DMX Fixtures, DMX Controller, DMX Converter, Virtual Lighting Panel, JNIOR Configuration.

JNIOR CONTROLLER

The JNIOR is a networked controller which can be considered to be a member of the *Internet of Things* (IoT). It was developed long before these terms and classifications were coined. It was intended to be, and successfully achieves the role of an inexpensive alternative to more elaborate and costly industrial control systems. This format of device has been available in a series of successive backwards compatible models for over 15 years. It provides integrators with a consistent and reliable source of controller hardware.

When connected to the network a remote system (or browser) can control a number of dry contact relay outputs and read the status of a set of digital inputs. A number of expansion modules are available providing additional relay outputs capable of switching power circuits; adding analog capabilities for 4-20ma and 10V input and output signals; providing for dimmer control of LED strips including RGB color; and for handling any local user interaction. The JNIOR also supports serial channels for communicating with other equipment.

The term JNIOR is an acronym which stands for Java Network Input Output Resource [1]. It can be pronounced as "junior" although the name is not intended to diminish the device's role in the world of controllers. In fact, beyond simply providing remote I/O resources, the JNIOR is programmable and is capable of providing sophisticated distributed control logic and data monitoring. It can perform autonomously in the absence of a network entirely. There are many applications where the JNIOR's network connection is used only for initial configuration. This controller can also be found embedded in other products where control logic and a network footprint are beneficial.

JANOS OPERATING SYSTEM

The JNIOR comes with its own *Operating System*. This is a complete preemptive multitasking operating system with all of the generic functionality found in multiuser computing systems. This includes a full network stack with secure protocols and a fully functional web server with server-side scripting capabilities.

The JANOS Operating System has been completely developed by INTEG and specifically for use with the JNIOR. This has allowed the system to be optimized for its function without extraneous overhead. It allows the inexpensive microcontroller at the heart of the JNIOR to perform the tasks of more complicated computing platforms.

Being fully developed by INTEG this OS contains no third-party source code. The advantage here is that when issues are identified, INTEG is 100% in a position to resolve the problem. In fact, many issues can be corrected the same day that they are identified. There is no one else to blame and issues cannot be "elevated". There is no outside group wherein issues become subject to separate prioritization, independent release schedules, and code obsolescence. Issues just get resolved.

Perhaps as important is the ability to easily expand functionality. That is the case here, where the capacity to generate the standard DMX512 signals was added. JANOS in the future will support that protocol as a core function offering the capabilities explored in this paper as built-in features.

The term JANOS is also an acronym. It actually is an acronym that incorporates an acronym. It stands for the JNIOR Automation Network Operating System. The name references an ancient Roman religious myth [2]. *Janus* is the god of beginnings, endings and time. This leads to an association with doorways, gates, and transitions. He is the god who oversees all comings and goings. One can imagine a relationship here to an I/O controller and that forms the basis of the JANOS operating system name.

APPLICATION PROGRAMMING

The JNIOR out of the box is fully functional as a remote I/O device. It can be used without running any *Application Program*. An application program however gives the user the ability to add unique and custom capabilities to the device. INTEG provides application programs such as *cinema.jar* which creates an environment specific to Digital Cinema where cues and other signals trigger definable macros.

Since most of the work is handled by the JANOS operating system, an application program need not be very complex. In the context of this paper we will see how a program containing a simple loop can handle the DMX512 universe. You can do almost anything with an application program. Providing documentation to support that flexibility is somewhat difficult. INTEG is therefore willing to freely support your programming efforts. All you have to do is contact us and let us work with you.

Since the introduction of the Series 4 JNIOR, application programming has been greatly simplified. These programs are written in standard Java and any of the available compiler tools can be used. You need only generate a JAR file. We are most familiar with *Netbeans* where there are a couple of simple configuration steps that insure that your project is built against the proper *JanosClasses.jar* runtime environment. Your program needs to be compiled to run under this specific runtime and to not reference any standard classes as might be supplied for programs targeted to run on PCs.

The *JanosClasses.jar* runtime file can be retrieved from the JNIOR as it resides in the */etc* folder. This library is JANOS version specific and programs compiled against one version are guaranteed to be compatible with later versions. A version of this JAR file is available that contains Javadoc and source code references in addition to the set of runtime classes.

HARDWARE REQUIREMENTS FOR DMX512

A DMX512 connection complies with the standard for RS-485 communications [3]. The JNIOR Model 410 AUX port supports RS-485 directly whereas the 412 and 414 models do not. You therefore need to use a 410 for DMX. The AUX port will be configured for RS-485 communications.

The standard DMX512 OUT connector is XLR type 5-pin female. The DMX512 standard [4] starting in 2004 prohibits the use of the 3-pin XLR connectors although these remain prevalent in the market. One issue with the 3-pin connections appears to be the polarity. Many devices supporting the 3-pin XLR connectors also provide a polarity switch. Here we will describe the proper 5-pin connection in compliance with the standard.

The JNIOR AUX output uses a DB9F connector as is common for serial communications. For the DMX512 implementation on the Model 410 you will need an adapter. This is relatively simple to construct. The connections are listed in Table I.

TABLE I
ADAPTER PIN CONNECTIONS

	5-pin XLR Female	DB-9 Male
Signal Ground (GND)	1	5
Data- (D-)	2	2
Data+ (D+)	3	8
Not Used (NC)	4, 5	1, 3, 4, 6, 7, 9

The DMX512 standard specifies the use of cable with a 120 Ohm characteristic impedance and 12 pF/ft. capacitance. These specifications are meant to insure that the maximum number of fixtures can be controlled (32) and be placed along the maximum length of wiring. For an adapter and for a simple DMX control arrangement the cable requirement is not that critical.

You can purchase a DMX512 XLR cable with a 5-pin female connector on one end and tinned bare wires on the other. Since most DB9M solder cup connectors take up to 20AWG cable you need to be careful about wire diameter or soldering becomes difficult. There are 18AWG DMX cables available. Those are too large (diameter of the wire is too big). The DMX512 standard calls out 24AWG cable (larger numbers are smaller diameters). That would be very easily soldered into the DB9M. Don't forget a cable hood to make things look professional.

You can use the 3-pin XLR if you want to. It is prohibited by the standard but that doesn't appear to deter everyone. The pin numbers are consistent (1, 2, and 3). You may run into the polarity issue and in that case might have to swap two of the wires in your adapter.

Note that your adapter should pass the cable shield through. Again, this is not critical in a simple arrangement but may be necessary in a complex stage setting where electrical noise is prevalent.

DMX512 VARIANCES

The Model 410 with properly constructed adapter can control a full DMX512 Universe (512 8-bit analog channels). One important note here is that this DMX512 port **is not isolated**.

In any facility, equipment connected to different A/C line circuits can have dramatically different ground potentials. When these devices are interconnected:

- Current may flow in ground (GND) circuits causing noise, possible product damage and potential hazards.
- *Common Mode Voltage* differences can place signals near or beyond operational limits preventing operation or reliable function.
- The likelihood of damage due to lightning or ESD events increases.

The DMX512 standard describes *isolated* connections which are able to maintain or at least restore operation even when common mode voltages exceed 1000s of volts. The Model 410 AUX port is not isolated. Differences in ground potentials can prevent operation and possibly damage the JNIOR. This is not a significant risk when the JNIOR and the lights being controlled are in the same vicinity and powered from the same A/C circuits.

The DMX512 standard specifies a maximum update rate of 44 samples per second. If the 512 channel message is repeated continuously with minimum timing per the specifications the maximum number of cycles is about 44. In reality not all DMX controllers can achieve that rate. The simple DMX application program that we will describe here comes close. It likely updates 42-43 times a second. If the complexity of the program is modified this number may change although the JNIOR serial I/O is interrupt driven and if programs are written carefully there are plenty of CPU cycles left to do other work.

Also you do have the option of reducing the size of the universe. A smaller number of channels can be transmitted. Dropping the stream to 256 channels for instance increases the update rate to 88/sec. Generally these update rates are far higher than necessary for good lighting control. Fixtures that implement motion often implement the actual movement from starting point to finish internally. So the rate or the smoothness of the action is not dependent on the DMX update rate or the controller.

Operating System Requirements

Each Series 4 JNIOR runs the JANOS Operating System. INTEG releases new versions of JANOS periodically. Generally issues are corrected with each release. Typically each includes a new set of features. Every new release is fully compatible with those before. Customers need not worry about an application interruption due to a change in operating system.

At this time the current JANOS release is v1.6.1 and v1.6.2 is now internally a *Release Candidate*. The latter is under test pending an upcoming release date. **JANOS v1.6.2 is required for this DMX application.** Customers can obtain and use release candidate OS and even Beta OS versions. The one requirement being that those units **must** be updated once the OS is released. INTEG only supports the most recent OS release. That means that if you have an issue our most likely first request is that you update to the latest OS and verify that issue remains a problem.

The update process is simple. First you copy a supplied UPD file to the */temp* folder using FTP or by dragging and dropping the UDP file in the Dynamic Configuration Pages (DCP) tool open under the browser. Then execute the *jrupdate –fup temp/(filename)* command from within the Command Console accessible with through Telnet or the DCP tool.

JNIORs can also be updated using the *Support Tool* (ST) which is an application available for use under Windows on the PC. In that case you need only open and run a supplied Update Project. An update project can be created to update not only the operating system but also application programs and to make changes to JNIOR configuration via the Registry.

In order to implement the DMX512 port, there have been a couple of modifications to JANOS. These are available as of v1.6.2 and you will need to update to that version to successfully run DMX control. Specifically the 250 KBaud setting required by DMX512 has been added to the serial port baud rate selection.

The DMX protocol also uses a Mark-After-Break signaling technique for synchronization and to indicate transmission of the start code and first channel. This signaling is not easily achieved with normal serial ports and especially those running under interrupt control. Special methods have been added to the program class for the AUX port allowing it to handle this aspect of the protocol.

Streaming the Channels

As we discussed previously, an Application Program is used to add new functionality to the JNIOR. Here we will implement a program called *dmx.jar* which will be responsible for generating the ongoing DMX512 data stream.

An example of the *dmx.jar* program is shown in Figure 1. The program needs to send the 512 channels repeatedly. Each channel is a byte, 8 bits. The DMX512 protocol defines a Mark-After-Break preamble which is followed by a single byte defining a *Start Code*. The default Start Code for our purposes is a 0x00. So we need to issue the preamble, send 513 bytes, and repeat. This is entirely performed by Lines 61-64.

The hidden complexity here is taken care of by JANOS. The serial ports are interrupt-driven. When a program writes data it is buffered and sequenced out while the program statements are allowed to continue. In Line 62 the MarkAfterBreak() method not only creates the proper 100ns Break condition followed by a 12ns Marking condition but it knows not to do so until all previous data has been transmitted (buffers are empty). So the program sleeps until it can perform the action.

The following write() statement in Line 63 executes immediately after the MarkAfterBreak preamble begins. This outputs a 513 byte data array. Each byte goes into the output buffer and, again, JANOS knows not to start transmitting serial data from the buffer until any active Mark-After-Break signal has completed. The two statements can be executed in this tight loop to create the proper output stream. Meanwhile the processor spends a lot time waiting and making itself available for other activities.

We create a separate thread here (Lines 43-73) whose sole purpose is to configure the AUX port properly and then spew out the continuous DMX512 stream. Lines 50 and 51 create an object which gives us access to the AUX port and opens that port. Lines 53 and 54 set the proper baud rate and data format for DMX512.

```
 1  package dmx;
 2
 3  import com.integpg.comm.AUXSerialPort;
 4  import com.integpg.comm.SerialOutputStream;
 5  import com.integpg.system.MessagePump;
 6  import com.integpg.system.SystemMsg;
 7
 8  public class Dmx {
 9
10      public static void main(String[] args) throws Throwable {
11
12          DMXport dmx = new DMXport();
13          Thread t = new Thread(dmx);
14          t.start();
15
16          MessagePump pump = new MessagePump();
17          pump.open();
18
19          for (;;) {
20              SystemMsg msg = pump.getMessage(1400);
21              String text = new String(msg.msg);
22              int pos = text.indexOf(",");
23              byte chan = (byte) Integer.parseInt(text.substring(0, pos));
24              byte val = (byte) Integer.parseInt(text.substring(pos + 1));
25
26              if (chan > 0 && chan <= 512)
27                  dmx.data[chan] = val;
28              else {
29                  for (int n = 1; n < 513; n++) {
30                      if (dmx.data[n] != 0)
31                      {
32                          String vals = n + "," + (dmx.data[n] & 0xff);
33                          msg = new SystemMsg();
34                          msg.type = 1400;
35                          msg.msg = vals.getBytes();
36                          pump.postMessage(msg);
37                      }
38                  }
39              }
40          }
41      }
42
43      private static class DMXport implements Runnable {
44
45          public byte[] data = new byte[513];
46
47          @Override
48          public void run() {
49              try {
50                  AUXSerialPort aux = new AUXSerialPort();
51                  aux.open();
52
53                  aux.setSerialPortParams(AUXSerialPort.SPEED_250000,
54                          AUXSerialPort.DATABITS_8, AUXSerialPort.STOPBITS_2, AUXSerialPort.PARITY_NONE);
55                  aux.setRS485(true);
56                  aux.enableReceivers(false);
57                  aux.enableDrivers(true);
58
59                  SerialOutputStream auxout = aux.getOutputStream();
60
61                  for (;;) {
62                      aux.sendMarkAfterBreak(100, 12);
63                      auxout.write(data, 0, data.length);
64                  }
65              }
66              catch (Throwable t) {
67                  System.out.println(t);
68              }
69          }
70
71      }
72
73  }
74
```

FIGURE 1
JAVA APPLICATION PROGRAM TO GENERATE THE DMX512 STREAM

Lines 55-57 establish the use of RS-485 and the unidirectional aspect of DMX512. In this protocol our transmitter is always active and the receiver has no purpose. We don't want flood an unused input buffer with all of our outgoing data and so we disable the receiver. The described adapter actually does not loopback to the receiver input anyway. Line 59 gives us access to the output stream used by the loop.

That's it. This thread is started by the main program in Lines 12-14. The *dmx.jar* program can be started from the Command Console just by typing 'dmx' and enter. You can set a Run key in the JNIOR's Registry that will start the program on boot for you. The program can be enhanced with a Watchdog capability that will restart it if for any reason execution is terminated.

CHANGING CHANNEL VALUES

The remainder of the main method in the application program of Figure 1 is one approach to controlling channel data. The DMXport thread class defines a byte[] called *data*. The array contains 513 bytes. The first being the protocol's Start Code which should remain unmodified with the initial value of 0x00. This is to always indicate that channel data follows in the protocol message.

To change the value of any DMX channel the program needs to simply write the byte associated with that channel to this array. Line 27 in the program updates a channel with a new value by referencing the data array in the DMXport class.

There are many ways to get new channel values. An enumeration is beyond the scope of this paper. Here we demonstrate one possibility using the inter-process communications capability in JANOS.

The program Lines 16 and 17 open a *Message Pump*. JANOS supports a messaging service to pass information from one process to another or to broadcast messages to all processes. The Message Pump object created here taps the message loop. A program can use that class to examine each and every message. It then has the responsibility of reposting messages not addressed to it or those that are broadcasted. Here in Line 20 we use a method that waits for and collects only the specific message addressed for this program. All other messages are properly handled for us in the background.

Messages have a type defined by a number 0 through 65535. Message numbers under 1024 are reserved for system use. There are a number of predefined system messages. Message numbers above and including 1024 may be assigned by the user. Here we picked 1400 as the DMX communications type. When another process posts a message with that type number, Line 20 picks it up for us.

The format of the message is user defined. Here in Lines 21-24 we convert the message content to a String and parse out the referenced channel number and its new value. Line 27 can then simply update the channel array which in turn alters the ongoing DMX stream.

The program provides one feature that will help the remote source of value changes (a simulated lighting panel) initialize its display. If a message is received with an invalid channel number, the program generates a series of responses using the same message number (1400) one for each non-zero channel. These are used at the other end to initialize the fader positions.

SIMULATED LIGHTING PANEL

Using HTML5 a simulation of a lighting panel with faders can be easily achieved. The description of this is beyond the scope of this paper. Any layout of faders along with other controls can be created. JavaScript is used to handle the events when faders are adjusted or controls activated. A simplified page allowing for the adjustment of the first 12 DMX channels could be displayed as shown in Figure 2.

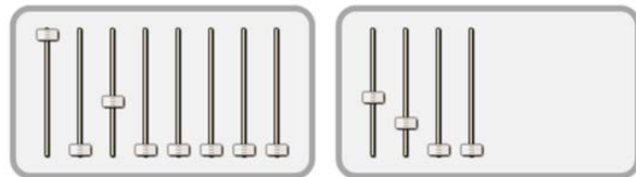

FIGURE 2
EXAMPLE WEB PAGE CONTENT

JANOS supports a robust built-in Websocket [5] interface which uses JSON [6] formatting to pass requests and receive responses. WebServers including that in JANOS allow an HTTP protocol connection to be upgraded to Websocket protocol. This is a seamless operation. Under JANOS Websocket connections may also by processed by application programs. They support authentication and can be secured under TLS v1.2. Here we use the built-in service for the DMX512 test implementation.

This interface can be used to handle everything from configuration to control. In this application when a fader is adjusted the associated JavaScript uses the Websocket mechanism to have JANOS issue an inter-process message of type 1400. So the browser formats the message and has it passed through to the running *dmx.jar* application program.

This results in a functional lighting panel simulation. A change in fader position creates in an immediate change in the brightness of the connected LED lamp. There is no perceivable delay.

SUMMARY

The Model 410 JNIOR supports RS-485 communications and beginning with JANOS v1.6.2, a DMX512 data stream can be properly generated. A simple adapter can be made to connect standard DMX cabling to the unit's AUX port. The JNIOR can then successfully control a full DMX512 Universe.

A small application program is required to generate the ongoing DMX512 stream. This can be customized to accept updated channel data as required by a user's application. This program can be set to run on boot and thus turn the Model 410 into a fulltime DMX controller.

REFERENCES

[1] Java™ is a trademark of the Oracle Corporation and its related entities.

[2] "Janus" god of beginnings, Wikipedia, Encyclopedia Britannica

[3] RS-485, TIA-485(-A), EIA-485 Standards for Electrical Characteristics of Generators and Receivers for Use in Balanced Digital Multipoint Systems, Electronic Industries Association

[4] DMX512-A Entertainment Technology, E1.11-2008 (R2013) American National Standard, Asynchronous Serial Digital Data Transmission Standard for Controlling Lighting Equipment and Accessories.

[5] Websocket Protocol, RFC 6455 Internet Engineering Task Force (IETF) Standards Track, tools.Ietf.org/html/rfc6455

[6] JSON JavaScript Object Notation, www.json.org

CONTACT INFORMATION

INTEG Process Group, Inc.,
2919 E Hardies Rd 1st Floor
Gibsonia, PA 15044
USA

724-933-9350
www.integpg.com