DYNAMIC DEVICE DISCOVERY

AMX Corporation

Duet Dynamic Device Discovery:

Technology Overview

TABLE OF CONTENTS

1. INTRODUCTION 3

1.1 PROGRAMMING IN CONCRETE 3
1.2 PROGRAMMING IN MOTION 3

2. PROGRAMMING USING DYNAMIC DEVICE DISCOVERY 4

2.1 DEFINITIONS 4
2.2 DDD FLAVORS 5
2.2.1 Static Binding 5
2.2.2 Dynamic Binding 5
2.3 THE NETLINX PROGRAMMING INTERFACE 6
2.3.1 Static Binding 6
2.3.2 Dynamic Binding 7

3. RUN-TIME BINDING 8

3.1 AUTOMATIC VS. MANUAL BINDING 8
3.2 BINDING PERSISTENCE 8

4. RUN-TIME DYNAMIC DEVICE DISCOVERY 10

4.1 SERIAL DEVICE DETECTION 10
4.2 USER DEFINED DEVICE DETECTION 10
4.3 IP DEVICE DETECTION 10
4.4 DUET MODULE DOWNLOAD AND RESOLUTION 10

5. DEVICE SDK CLASSES AND CONSTANTS 12

1. Introduction

This document describes Duet Dynamic Device Discovery (DDD); the process of detecting new devices within a NetLinx/Duet system, binding these devices to application instances and starting a Duet module to control the new device. The purpose of this document is to present the DDD terms and definitions as well as some of the scenarios in which DDD could be utilized.

1.1 Programming in Concrete

In the current NetLinx/Duet programming environment, the developer must know exactly what devices will be connected to the system, what physical interfaces these devices will be connected to and what control/protocol module is to be used to communicate with the device (be it NetLinx or Duet Module). As is obvious, this is rather limiting. If the device for whatever reason needs to be moved to a different physical interface, for example a different serial port, the NetLinx application program must be modified, re-compiled, downloaded to the master and the master rebooted. By "application program", we refer to the developer defined NetLinx program that runs on a master and controls a system. Or for example, a physical device breaks and must be swapped out with a comparable, yet different device that talks a different protocol. The application would need to be re-compiled with a different control/protocol module in order to talk to the new device. At an off-site installation without network access, this may require an inconvenient and expense service call.

1.2 Programming in Motion

Dynamic Device Discovery (DDD) was created to take advantage of Java's Dynamic Class Loading and the Duet Standard NetLinx API (SNAPI). Java loads each class only as it is needed. Therefore it is feasible to load a Duet device control module on the fly as each new device is discovered. SNAPI provides a fixed interface for communicating with a certain type of device from the NetLinx application program. Take for example a VCR. The majority of control features are common to all VCRs (play, stop, pause, etc.). SNAPI provides the application developer the ability to write common code that will control any type of VCR having an associated Duet module. The underlying Duet module could be swapped in and out based on the actual physical device with no changes needed to the higher-level application.

2. Programming Using Dynamic Device Discovery

In the DDD world, all of the existing capabilities of NetLinx and Duet remain. For those portions of the system that will never change, static programming using DEFINE_MODULE continues to provide simple, modular control of a peripheral device. But with DDD the developer now has the ability to program for the future.

2.1 Definitions

DDD introduces several new concepts:

- **APPLICATION DEVICE**

A Duet virtual device (41000-42000) that is used as a control interface to a physical device. All control requests are made to the application device rather than the physical device.

- **BINDING**

In static programming, the application device is forever associated with the NetLinx physical device. In DDD, this association is dynamic. The act of associating an application device with a physical device is called 'binding". The different types of binding are discussed in Section TBD.

- **DEVICE DISCOVERY**

In DDD, physical devices are detected in the system at run-time. There are currently two different methods of detection; via Dynamic Device Discovery Protocol (DDDP) or user defined via the master's Web interface. The different methods of device discovery are discussed in Section TBD.

- **SDK CLASS**

Each application device in the DDD world is associated with a particular device type as defined by SNAPI. Example device types are VCR, DVD and Receiver. Each of these device types corresponds with a Java Interface within the Duet Device Software Development Kit (DeviceSDK). When writing programs for DDD, the application developer specifies the device type of a particular application device using one of these SDK Class names. The current list of class names and their associated NetLinx convenience constants are listed in Section TBD.

- **POLLING**

Dynamic physical devices can be detected by DDDP through both serial and IP interfaces. IP connections are able to utilize the network's higher layers of multicast to announce their existence, while serial devices speak a fixed protocol that is incompatible with DDDP. Serial devices are passive and will only announce their existence if polled to do so. Therefore the application program developer must specify which NetLinx interfaces/ports (i.e. serial ports) should be polled for serial devices.

2.2 DDD Flavors

In DDD, the device discovery activity is always dynamic. That is, devices will always be detected at run-time. But DDD splits the binding activity into two different categories, program-defined binding, also known as "static binding", and run-time defined binding, known as "dynamic binding".

2.2.1 Static Binding

With static binding, the developer specifies a permanent binding between an application device and a physical port, for example, a particular serial or IR port. At run-time, any device detected on that port is automatically associated with the designated application device. This binding type would be used when the developer wants to hardcode what port is used for a device, but does not know what manufacturer's device will actually be connected. Static binding is not available for IP connected devices, since the IP address value of a device is subject to change due to IP network topology. For example, if DHCP were enabled for the peripheral device, a hard-coded IP address within the NetLinx application would be inadequate due to the nature of dynamically acquired DHCP IP addresses. Only actual NetLinx D:P:S values are allowed for static binding of physical ports.

2.2.2 Dynamic Binding

With dynamic binding, the application device and the physical port are completely disassociated programmatically. The developer defines the application devices and their associated SDK class but makes no specification as to what physical port they are bound to. As devices are discovered at run-time, the new physical devices are bound to an application device either automatically, or manually via the master's web server. Auto binding is the only dynamic binding option available for IP connected peripheral device due to dynamic nature of IP addresses as discussed earlier. Auto binding may be enabled or disabled through the master's web server.

2.3 The NetLinx Programming Interface

Just as the current static programming method utilizes DEFINE_MODULE to define its peripheral control, DDD introduces additional NetLinx APIs for the definition of dynamic facilities.

2.3.1 Static Binding

The STATIC_PORTL_BINDING API designates an application device along with its SDK class and the physical interface it is bound to. The complete API is:

```
STATIC_PORT_BINDING (DEV duetDevice, DEV netlinxDevice, char[] deviceType,
                     char[] friendlyName, integer polled)
```

The duetDevice parameter is a NetLinx DEV object with a device value in the range of 41000-42000. This is the virtual device that the NetLinx program uses to interact with the physical device. NetLinx events from and to the device through SNAPI will occur on this device.

The netlinxDevice parameter specifies the physical device port interface. For example, on an NI-3000 serial port 1, this value would typically be 5001:1:0. Any device discovered on this physical port will be automatically bound to the specified duetDevice.

The deviceType parameter is a string value containing the SNAPI device SDK class name that is associated with the duetDevice. For programmer convenience, constants are provided for each SDK type available. All constants have the format DUET_DEV_TYPE_*xxxx* where *xxxx* is the device type. For example, DUET_DEV_TYPE_VCR is a constant for the value "com.amx.duet.devicesdk.VCR". While either the constant or the long-hand version of the string will work, use of the constants will be less prone to error as any invalid value within the string will prevent the Duet module from loading.

The friendlyName is a string description of the device that will appear on the master's Web interface. For example, "Living Room VCR". The friendly name makes it easier for the installer to distinguish different devices. "41000:1:0" may have no meaning to the installer, but "Conf DVD" might.

The polled parameter indicates whether or not the specified netlinxDevice should be polled for dynamic device detection. For programmer convenience, constants are provided; DEVICE_NOT_POLLED (0) indicates not polled, DUET_DEV_POLLED (1) indicates polled. Serial devices are polled, any other device type is not. Specifying a serial device port as not polled should not be done if the application programmer only wants device discovery to occur for that port via the master's web interface (i.e. manually entered).

2.3.2 Dynamic Binding

The dynamic binding capability is accomplished through two new NetLinx APIs:

DYNAMIC_APPLICATION_DEVICE (DEV duetDevice, char[] deviceType, char[] friendlyName)

DYNAMIC_POLLED_PORT (DEV netlinxDevice)

DYNAMIC_APPLICATION_DEVICE specifies a Duet device that is completely dynamic. A dynamically discovered device matching the specified deviceType could be bound to the duetDevice from anywhere in the system.

DYNAMIC_POLLED_PORT designates a NetLinx serial port that should be polled for dynamic device detection. This API must be called for each serial port that can dynamically have a device plugged into it.

Utilizing these two APIs, a developer can create a list of application devices and configure DDD to detect devices on select serial ports. In addition, dynamic device detection of IP devices and Web based user-defined devices are available. Any dynamically discovered device detected from any of these sources could potentially be bound to a dynamic application device.

3. Run-time Binding

3.1 Automatic vs. Manual Binding

Through the master's web server interface, the master can be configured to automatically determine what Dynamic Application Device a newly discovered Dynamic Physical Device should be bound to. The use of this capability requires that a newly discovered physical device have only one clear choice of application device. The application device cannot already be bound to another physical device. In other words, the application device must be "orphaned" awaiting a binding. The decision to bind is based on the application device's SDK class. For example, if there is only one orphan DUET_DEV_TYPE_VCR in the system and a new VCR is detected, the two entities will be automatically bound and an associated control module will be started. This capability provides a complete hands-free approach to installation.

If auto-binding is enabled but the master cannot determine which application device to bind, the conflict must be resolved via manual binding through the master's web interface. For example, if there are two orphan DUET_DEV_TYPE_VCRs in the system and a new VCR is detected, no binding will occur. Using the master's web interface, the installer must select which orphan application device to bind to the new physical device.

Automatic binding based on orphan application devices is dynamic. In the two-VCR example from above, as soon as the installer manually binds one of the application devices with a physical device, the remaining DUET_DEV_TYPE_VCR application device can be automatically bound to the last physical VCR because there is now a clear one-to-one correspondence.

Automatic binding can be turned off. If disabled, each binding of an orphan application device to an orphan physical device must be manually executed via the master's Web interface.

Static bindings, as defined in the NetLinx program via STATIC_PORT_BINDING() calls, are not affected by automatic or manual binding.

3.2 Binding Persistence

Bindings persist between reboots. The only way a binding can be removed is if:

a) The binding is manually unbound via the master's web interface

b) A new/different device is detected on the bound physical port that does not currently have a control module active.

The first case is self-explanatory but can have some undesired side effects. For example, if a running physical device is bound to an application device and the binding is destroyed via the web interface, the existing binding will be broken and any running control module will be unloaded. But because the physical device is still attached, it could be dynamically detected again and possibly automatically bound back with the same orphan application device that was just made available via the unbinding process. Because of this, it is advised to **always disable auto-bind prior to a manual unbind via the web interface.**

The second case of removing a binding occurs when a different device is detected on an existing, bound physical port. The one stipulation is that the existing binding cannot currently have a running control module. This could occur either immediately after a reboot when the existing binding's physical device has not been re-acquired or if a running control module terminates itself due to lack of communication with its physical device. This case would occur for example, when a physical device is hot-swapped. When the old device is removed and its associated control module is terminated due to lack of communication, the binding can be automatically removed when the new device is detected.

Bindings with active control modules have precedence and can only be destroyed manually via the Web interface.

Duet provides a mechanism, accessible on the master web interface, for purging all device bindings and modules on subsequent reboot of the master.

4. Run-time Dynamic Device Discovery

4.1 Serial Device Detection

As mentioned earlier, serial devices are polled. Periodically, DDD will send a fixed string out the serial port. When a DDDP-enabled device is attached to the serial port it will respond with a DDDP beacon response message, containing property information about the connected device. This will initiate a chain reaction including termination of polling on that port, binding of the application device to the physical port, and Duet module activation.

If a running Duet module determines that communication with the device has been lost, it will be terminated and DDD will return to polling the serial port.

4.2 User Defined Device Detection

For devices that do not support DDDP (serial and IP) and devices that have no mechanism for transmitting data (e.g., IR devices), the device property information can be entered via the master's web interface. The information entered for the device will be used to find the appropriated Duet module. Therefore knowledge of the required match values is required. All match values are case sensitive.

4.3 IP Device Detection

IP device detection in DDD is accomplished through the receipt of a multicast beacon message from the IP device. Because multicast spans an entire network, every Duet master in a networked multi-master system will detect a new device. When one master binds to the device, it broadcasts ownership of the device to all of the other masters. If auto-bind is enabled on multiple masters and more than one master has an application device matching the IP device type, a race condition can occur with multiple entities trying to take control of the same device. This can be avoided by carefully architecting the system to prevent these types of conflicts, but in many cases this is not feasible.

One solution is to turn auto-bind off on all masters that could have a conflict. Bindings on these masters would need to be done manually via their Web control.

In order to reduce these conflicts and still allow each master to auto-bind, the masters can be configured to only detect those devices that are on their own subnet. In this way, each master could be configured under a different subnet, thereby establishing clear ownership of all IP devices added to its subnet.

4.4 Duet Module Download and Resolution

In order to complete a bind activity, the master's firmware must load and start the appropriate Duet module to control the new device. In the static programming world, these modules are bundled and downloaded with the NetLinx application. But with Dynamic Device Discovery, the proper Duet modules are not known at program time and thus cannot be part of the programming download.

Currently, Dynamic Device Discovery requires that the necessary modules must be resident on the master. The capability has been added to the master's web control to download Duet module .jar files from any networked PC.

Due to Java's dynamic class loading, it is not safe to swap out a .jar file at run-time, since it could cause class definition conflicts that generate exceptions and termination of the Duet module. To avoid this, the downloaded .jar files are placed in an "unbound" holding directory on the master. As each module is started by the master, it is first copied into a run-time "bound" directory to prevent overwrites by subsequent downloads.

Each active binding requires the loading and instantiation of an associated Duet Module. To determine the correct module to load, the master's firmware uses a set of property values to determine the best possible match. When a new device is detected in the system, it provides this set of property values. The master firmware then searches through its list of modules looking for the one that best matches the new device's property values. Once a Duet module has been associated with a binding, that association is persistent throughout the life of that binding as long as the run-time instance of the module exists.

This persistent association can present problems whenever a new version of the module is downloaded. Because newly downloaded modules are placed in the holding directory, the new module will not be picked up until the old version in the run-time directory is deleted. But the run-time version cannot be deleted while the module is active. Due to this issue, the master web server now has the capability to enable purging of all run-time modules upon the next system reboot. When the master is rebooted with purge selected, all run-time versions of the Duet modules will be deleted prior to any Duet modules being activated. This will force the master to re-resolve each binding with its appropriate Duet module from the holding directory. This mechanism is used both to update existing modules as well as to purge old modules from the system for devices that no longer exist.

Rather than requiring the module be pre-resident on the master, module loading can be dynamic. The module may be retrieved from AMX's web site, from a device-specified URL, or even from the device itself. In all of these cases, the retrieved module will be placed in the run-time directory and persist between reboots, thereby eliminating the need to re-retrieve the module.

5. Device SDK classes and constants

The following is the current list of device SDK classes and their associated NetLinx Constant values. This list is subject to change as new device types are added to the SDK.

DUET_DEV_TYPE_AUDIO_CONFERENCER =	com.amx.duet.devicesdk.AudioConferencer
DUET_DEV_TYPE _AUDIO_MIXER =	com.amx.duet.devicesdk.AudioMixer
DUET_DEV_TYPE _AUDIO_PROCESSOR =	com.amx.duet.devicesdk.AudioProcessor
DUET_DEV_TYPE _AUDIO_TUNER_DEVICE =	com.amx.duet.devicesdk.AudioTunerDevice
DUET_DEV_TYPE _CAMERA =	com.amx.duet.devicesdk.Camera
DUET_DEV_TYPE _DIGITAL_MEDIA_PLAYER =	com.amx.duet.devicesdk.DigitalMediaPlayer
DUET_DEV_TYPE _DSS =	com.amx.duet.devicesdk.DigitalSatelliteSystem
DUET_DEV_TYPE _DVR =	com.amx.duet.devicesdk.DigitalVideoRecorder
DUET_DEV_TYPE _DISC_DEVICE =	com.amx.duet.devicesdk.DiscDevice
DUET_DEV_TYPE _DOCUMENT_CAMERA =	com.amx.duet.devicesdk.DocumentCamera
DUET_DEV_TYPE_AUDIO_TAPE =	com.amx.duet.devicesdk.AudioTape
DUET_DEV_TYPE _HVAC =	com.amx.duet.devicesdk.HVAC
DUET_DEV_TYPE _KEYPAD =	com.amx.duet.devicesdk.Keypad
DUET_DEV_TYPE _LIGHT =	com.amx.duet.devicesdk.Light
DUET_DEV_TYPE _MONITOR =	com.amx.duet.devicesdk.Monitor
DUET_DEV_TYPE _MOTOR =	com.amx.duet.devicesdk.Motor
DUET_DEV_TYPE _POOL_SPA =	com.amx.duet.devicesdk.PoolSpa
DUET_DEV_TYPE _PREAMP_SURROUND =	com.amx.duet.devicesdk.PreAmpSurroundSoundProcessor
DUET_DEV_TYPE _RECEIVER =	com.amx.duet.devicesdk.Receiver
DUET_DEV_TYPE _SECURITY_SYSTEM =	com.amx.duet.devicesdk.SecuritySystem
DUET_DEV_TYPE_SENSOR_DEVICE	com.amx.duet.devicesdk.SensorDevice
DUET_DEV_TYPE _SETTOP_BOX =	com.amx.duet.devicesdk.SettopBox
DUET_DEV_TYPE _SLIDE_PROJECTOR =	com.amx.duet.devicesdk. SlideProjector
DUET_DEV_TYPE _SWITCHER =	com.amx.duet.devicesdk.Switcher
DUET_DEV_TYPE _TV =	com.amx.duet.devicesdk.TV
DUET_DEV_TYPE _VCR =	com.amx.duet.devicesdk.VCR
DUET_DEV_TYPE _VIDEO_CONFERENCER =	com.amx.duet.devicesdk.VideoConferencer
DUET_DEV_TYPE _VIDEO_PROJECTOR =	com.amx.duet.devicesdk.VideoProjector
DUET_DEV_TYPE_VIDEO_WALL	com.amx.duet.devicesdk.VideoWall
DUET_DEV_TYPE _VOLUME_CONTROLLER =	com.amx.duet.devicesdk.VolumeController
DUET_DEV_TYPE _WEATHER =	com.amx.duet.devicesdk.Weather

3000 Research Drive • Richardson • Texas, 75082
Phone 800.222.0193 • www.amx.com