NexDome firmware for Arduino

This firmware was professionally developed for NexDome by Tigra Astronomy. The code is open-source and is licensed under the Tigra Astronomy MIT license.

Table of Contents

Development Environment

The Arduino IDE is great for casual experimentation but is rather limiting for serious software development. This project was developed using Microsoft Visual Studio, with the VisualMicro Arduino extension, which makes it easy to have *local libraries* and *shared code projects* and reference them in exactly the same way you would in a C# project, without having to install the library or move it to some aesoteric location.

Therefore, to compile this project, you will need Microsoft Visual Studio 2017 or later, and the VisualMicro Arduino extension. Both products have free versions that can be used to build this solution.

One useful feature of the Arduino IDE is that it includes the ability to compile and upload sketches to the target device in one operation. However, the process of installing the Arduino IDE, plus all of the libraries required, downloading the source code, compiling it and finally uploading the compiled sketch is too risky and complicated for non-technical end users. This also would have tied the project to the Arduino IDE, which would have restricted our development practices. We have therefore produced a firmware uploader utility written in C# within the same solution. This enables distribution of pre-compiled firmware in Intel Hex format (`*.hex`) and these files can be directly uploaded to the Arduino using a simple command.

Design Philosophy

Within the limitations of the Arduino platform (a resource-constrained embedded system), we have tried to apply the SOLID principles of object oriented design:

- `S` - Single responsibility principle; each class should have only one responsibility
- `O` - Open/Closed principle; open for extension, closed for modification
- `L` - Liskov substitution principle; superclasses and subclasses classes should be interchangeable
- `I` - Interface segregation principle
- `D` - Dependency inversion principle; depend on abstractions not details

Well-factored object oriented code that adheres to the SOLID principles should be loosely coupled, highly cohesive, testable and have low viscosity for future maintenance.

Due to severe resource constraints imposed by the target platform, we have had to compromise somewhat on this ideal and use a more procedural style in order to save memory and code space. Due to lack of a unit testing framework for the Arduino platform, we have also not produced unit tests.

Memory Management

Dynamic memory allocations have been agressively avoided. As a resource-constrained embedded system with just 2Kb of data memory, there is not much space available for a heap and we can't tolerate "Out Of Memory" errors at runtime. The system must be stable for days, months or even years at a time so the memory management strategy must be frugal, deterministic and stable.

Our solution to this is to statically pre-allocate as many objects as possible once, in global scope, then never delete them. The top level `.ino` sketch file contains these allocations either as statically initialized global variables or in the setup() method and this essentially forms the Composition Root for the system. The main exception to this rule is in the XBee state machine classes, which are created and destroyed as necessary - however the system will usually remain in the `Online` state so there will be a stable configuration most of the time.

Since we can assume that most objects are never freed, there is little to be gained from the use of smart pointers and we have chosen to avoid the overhead and use "raw" pointers where necessary. Where possible, we have used references (`&`) and pass-by-reference, and these references typically resolve to one of the objects defined in global scope in the main sketch file.

We make use of the C++ standard template library defined in namespace `std::` and provided by the library `ArduinoSTL` . This is perhaps unusual in the context of an Arduino sketch, for reasons we don't really understand. Perhaps many Arduino programmers are not familiar with the full capabilities fo th C++ language (a situation that the Arduino IDE and approach to programming seems to propagate). Perhaps it is because many people believe that these classes require more memory and the Arduino is somehow "not powerful enough" to use them. In fact, these classes are highly optimised and do not incur much, if any, overhead. What they do provide is algorithms and data structures that have been highly polished over many years, and a degree of readability and modularity to the code. We think that the trade-off of a slight memory and performance overhead is more than worth the extra readability and maintainability that it gives the code. We make good use of the `std::vector<T>` class to manage collections and in particular for sending, receiving and manipulating XBee API data frames. Typically we construct vectors with a reserved capacity, so that re-allocation of memory is not necessary, and we never free these structures so they remain permanently allocated (note: passing by reference does not copy the data). We also make use of `std::string` .

Motor Control

The stepper motors have a Direction/Step/Enable hardware interface and are driven by generating a square wave onto the Step pin while the Enable pin is asserted. Direction of movement is controlled, self evidently, by the Direction pin. Negating the Enable pin removes energy from the motor coils and releases holding torque.

The stepping process is time sensitive and demands high throughput speeds and consistent step pulse train generation for smooth motor operation. Tigra Astronomy has developed a control library for stepper motors that ensures that steps can be delivered at high speed and with consistent timing, regardless of what is happening in the Arduino main loop. The popular AccelStepper library does not achieve this because it relies on being called within the Arduino main loop and mingles all of the acceleration computations in with the step generation.

Here we see a clear advantage in paying attention to the Single Responsibility Principle. Tigra's stepper driver logically divides stepping into two parts:

Step Generator

A step generator (implements: `IStepGenerator`) is responsible for generating a pulse train where each rising edge causes the motor to make one step. The step generator is responsible for timing (step speed) and has no concept of position, direction or the type of steps (whole steps, microsteps, etc.)

We have provided a single implementation, `CounterTimer1StepGenerator`, which uses the Timer 1 block of the AVR processor to generate accurately timed pulses with 50% duty cycle. The timer is configured to generate interrupts using the `OCR1A` compare register. The timing source is the undivided system clock, which allows for a theoretical stepping bandwidth of about 244 steps/second up to 16,000,000 steps/second.

Step Sequencer

The step sequencer (implements: `IStepSequencer`) carries the responsibility of writing the correct hardware signals to the motor driver and keeping track of the step position.

Our `MicrosteppingMotor` class provides the `IStepSequencer` implementation and allows for acceleration and deceleration. `MicrosteppingMotor` also keeps track of the current step position and enforces limits of travel on the motors.

Acceleration

The `MicrosteppingMotor` class implements acceleration and deceleration based on the equation of uniform acceleration, $v = u + at$. This reduces the risk of stalling, especially when moving heavy loads.

The `ComputeAcceleratedVelocity` method is called once per Arduino main loop to recompute the motor velocity and acceleration curves. We have found that acceleration can be treated as a lower priority task and does not need to be computed for every step. This allows us to schedule it in the main loop with other tasks and maintain a clean separation of concerns in the code.

The "Ramp Time" (the time taken to accelerate from rest to maximum speed) is configurable by the user. Ramp time is specified in milliseconds. 250 to 500 milliseconds is usually sufficient for moderate loads but for more massive loads this can be increased.

Speed and Power Considerations

The NexDome controllers us a 15.3:1 gearing arrangement to drive both the rotator and shutter mechanisms. This arrangement provides a naturally stable system at rest which means that holding torque in the stepper motors is unnecessary. Therefore, the step drivers are disabled once motion has ceased. This reduces power consuption and keeps the motors and step drivers cool when not actively driving the mechanism.

While the AdvancedStepper code is capable of very high step rates, there is a trade-off between maximum stepping speed, available torque and power consumption and mechanical considerations inherent to the structure. The firmware provides commands for reading and writing the maximum step rate (`@VR` , `@VW`) and acceleration ramp time (`@AR` , `@AW`) so that the end user can manage this speed/torque/power tradeoff to suit local conditions. Conservative defaults are provided.

The Tigra `AdvancedStepper` code is capable of driving steps at a theoretical rate of at least 50,000 microsteps per second on an Arduino AVR family CPU (UNO, Leonardo). In practice the motors and driver modules used will dictate the maximum

possible speed beyond which the motors may be unable to provide sufficient torque and may stall. Sensible factory defaults have been carefully chosen and we do not advise adjusting the motor parameters away the factory defaults without good reason.

Command Processor

Command processing is handled by the `CommandProcessor` class. This is actually a port of a more object-oriented library (also developed by Tigra Astronomy) but is one area where we found it necessary to favour smaller resource usage over code structure. Therefore our object oriented design has been collapsed into more procedural code in this implementation.

When a well formed command is received from the serial communications channel it is passed to `DispatchCommand()` which decides whether it is a local or remote command. If local, it is passed on to `CommandProcessor::HandleCommand()`. Otherwise it is passed to teh XBee state machine for transmission on to the remote device.

Each command verb has its own handler method. `CommandProcessor::HandleCommand()` decides which handler method to call based on the command verb.

All command handlers return a `Response` structure, which contains the text (if any) to be transmitted back to the client application.

Command Protocol

Command Grammar

Commands have the form:

`@` `Verb` `target` `,` `Parameter` `<CR>` `<LF>`

The parts of this command syntax are:

- `@` is a literal character that marks the start of a new command and clears the receive buffer.
- `Verb` is the command verb, which normally consists of two characters. Single character verbs are also possible but in this case the entire command is a single character.
- `Device` is the target device for the command, generally `R` (rotator) or `S` (shutter).
- `,` is a literal character that separates the device ID from the parameter. The comma is required if a parameter is present and should be absent if there is no parameter.
- `Parameter` is usually a signed or unsigned decimal integer and is the data payload for commands that need data. Refer to the notes for each command for specific requirements.
- `<CR>` `<LF>` is the command terminator and submits the command to the command dispatcher. Only one is required and if both are present then they can be in any order.

Example: `@AWS,1000` {set shutter acceleration ramp time to 1000 ms}.

Errors

Any unrecognised or invalid command responds with the text `:Err#`.

Responses

In general, responses always begin with `:` and end with `#`.

Unless otherwise stated in the table below, all commands respond by echoing their command code and target device, in the format:

`:` cct `#`

where `cc` is the original command verb and `t` is the target device ('R' for rotator, 'S' for shutter).

The parameter value (if any) is not echoed. For example, the response to `@GAR,1000` is `:GAR#`. Receipt of this echo response indicates that the command is valid and was successfully received.

Commands that return a value include the value immediately after the target and before the terminating `#`. Example: `:VRR10000#`

Any command that cannot be processed for any reason will respond with `:Err#`

Other status and debug output may be generated at any time, not necessarily in response to any command. See Event Notifications below.

Command Details

In the following table, upper case letters are literal characters. Lower case letters are placeholders.

Cmd	Targets	Parameter	Response	Example	Description
AR	RS	none	:ARddddd#	@ARR	Read acceleration ramp time in milliseconds
AW	RS	ddddd	:AWt#	@AWS,1000	Write acceleration ramp time
FR	RS	none	:FRstring#	@FRR	Reads the semantic version (SemVer) string of the firmware.
GA	R	ddd	:GAR#	@GAR,180	Goto Azimuth (param: integer degrees)
HR	R	none	:HRRddddd#	@HRR	Home position Read (steps clockwise from true north)
HW	R	ddddd	:HWR#	@HWR,1000	Home position Write (seps clockwise from true north)
PR	RS	none	:PRt-dddd#	@PRR	Position Read - get current step position in whole steps (signed integer)
PW	RS	dddd	:PWt#	@PWR,-1000	Position Write (sync) - set step position
RR	RS	none	:RRtdddd	@RRS	Range Read - read the range of travel in whole steps.
RW	RS	ddddd	:RWt#	@RWR,64000	Range Write. Sets the maximum shutter travel or dome circumference, in steps.
SR	RS	none	:SEt,…#	@SRR	Requests an immediate status report. Status returned as comma separated values.
SW	RS	none	:SWt#	@SWS	Performs an immediate "hard stop" without decelerating.
VR	RS	none	:VRtddddd#	@VRR	Velocity read - motor speed in whole steps per second
VW	RS	ddddd	:VWt#	@VWS,10000	Velocity Write - motor speed in whole steps per second
ZD	RS	none	:ZDt#	@ZDR	Load factory defaults into working settings (does not save)
ZR	RS	none	:ZRt#	@ZRR	Load saved settings from EEPROM into memory

Cmd	Targets	Parameter	Response	Example	Description
ZW	RS	none	:ZWt#	@ZWR	Write working settings to EEPROM

Event Notifications

The firmware will produce notifications of various events. These may arrive at any time, for example in between a command and a response, but they will never divide a response. Client software must be prepared to handle these event notifications.

Format	Description
XB->state	The current state of the XBee communications link. The states are enumerated below.
XB->Start	Waiting for XBee to boot up or reboot
XB->WaitAT	Waiting for XBee to enter command mode
XB->Config	Sending configuration
XB->Detect	Attempting to detect remote device
XB->Online	Communications link established
Pddddd	Rotator position (ddddd = signed decimal integer)
Sddddd	Shutter position (ddddd = signed decimal integer)
:SER,p,a,c,h,d#	Rotator status report.
p = current azimuth position in whole steps,	
a = AtHome (1 = home sensor active, 0 = home sensor inactive)	
c = dome circumference in whole steps	
h = home position sensor location, in whole steps clockwise from true north	
d = dead zone (reserved for future use)	
:SES,p,o,c#	Shutter status report.
p=position in steps,	
o=state of open limit switch, 1=active, 0=inactive	
c=state of closed limit switch, 1=active, 0=inactive	
:left#	The rotator is about to move to the left (counter-clockwise)
:right#	The rotator is about to move to the right (clockwise)
:open#	The shutter is about to move towards open
:close#	The shutter is about to move towards closed

Note: position updates occur about every 250 milliseconds while motion is occurring. When motion ceases, an SER or SES status event is emitted and this indicates that motion of the corresponding motor has ceased.

Other Output

It is possible that other undocumented output may be produced and the host application should be constructed in such a way that this output is ignored. Diagnostic output should not be relied upon by a host application for any purpose.

Arduino Libraries Used

- ArduinoSTL - standard template library (install from library manager)
- eeprom - for reading and writing the nonvolatile storage (install from library manager)
- AdvancedStepper - Tigra Astronomy's advanced stepper motor control (included/local)
- XBeeAPI - NexDome specific, used for sending, receiving and parsing XBee API data frames (included/local)
- XBeeStateMachine - NexDome specific, used to control the sate of XBee communications (included/local)
- Timer - Tigra Astronomy's timer utility, used primarily for monitoring timeouts. (included/local)

Summary of XBee Operation

Key Requirements

The goal of the XBee implementation is to ensure a robust, error tolerant communication between the rotator module and the shutter module. The modules must be able to easily find each other and communicate, and to verify that communication has been successful, while avoiding any possibility of interfering with neighbouring installations.

From the factory, XBees are configured in a "transparent serial replacement" mode where any data sent to one is relayed to all others on the network. This mode doesn't really address any of the key aims, except that it works simply and without much configuration. In particular, it is likely to cause problems with multiple installations.

Proposed Method of Operation

We will assume that all of the devices in one dome are on the same PAN (Personal Area Network) and that devices in different domes will be on different PANs.

The rotator module will assume the role of Coordinator, default PAN ID 0x6FBF, with automatic PAN ID and channel reassignment. In this mode, the module will start up and scan for existing PANs and channels, and will choose a PAN and channel that i snot currently in use.

The Shutter module will assume the role of an Endpoint Device, default PAN ID 0x6FBF and will be configured to associate to the Coordinator with the strongest signal.

The XBees will be configured to operate in API mode (rather than transparent mode), so that the PAN can be monitored and communications will be fault-tolerant. 64-bit addressing will be used.

As part of the configuration process, the XBees will be reset to factory defaults and then all required parameters explicitly set. No pre-configuration is required.

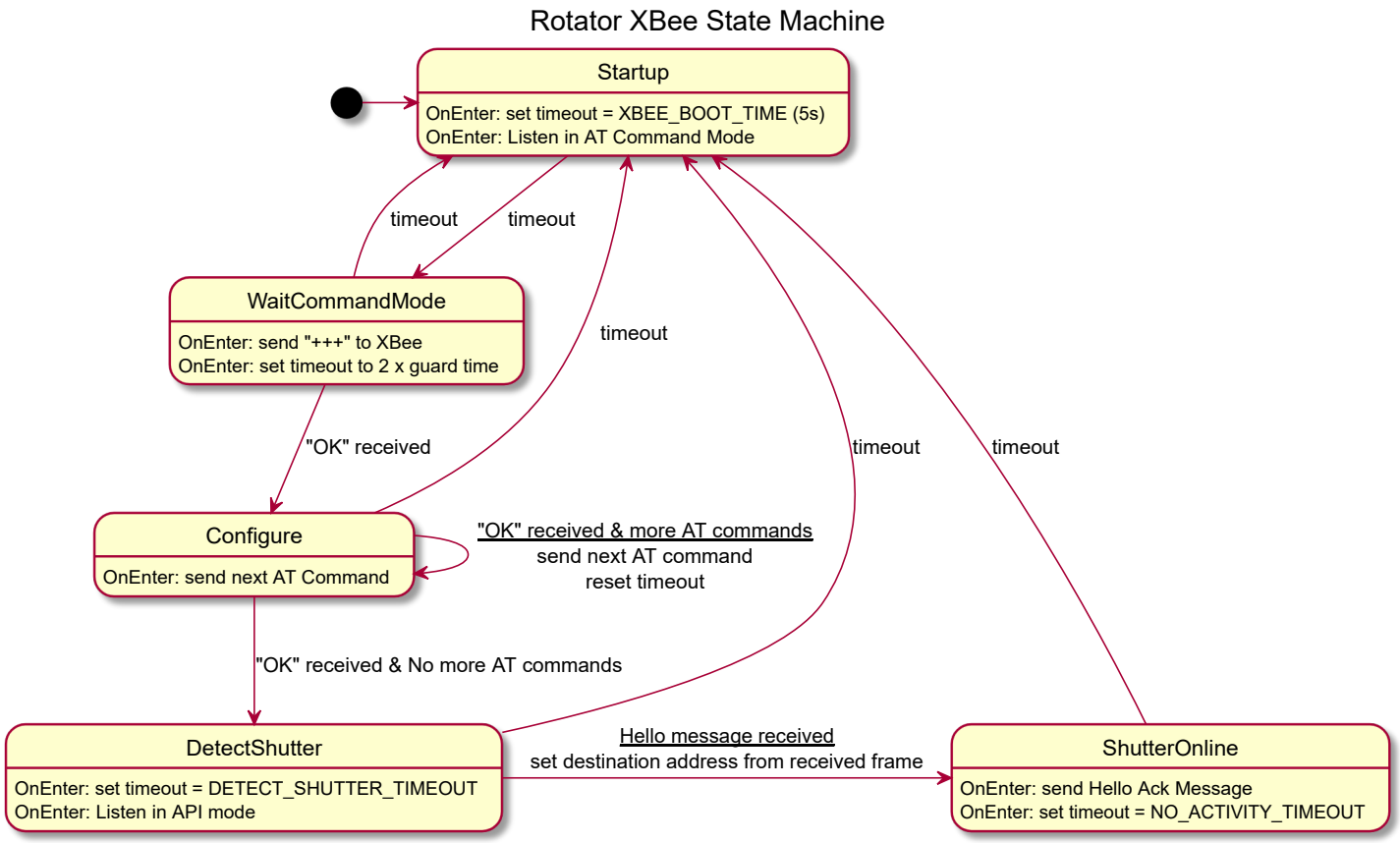
The XBee communications link will be managed by a state machine. If the communications link is lost for any reason, the state machine will automatically attempt to re-establish the link, rebooting and reconfiguring the XBees as needed. Whwnever there is a state change, this will be reported to the user via the serial port.

XBee State Machine

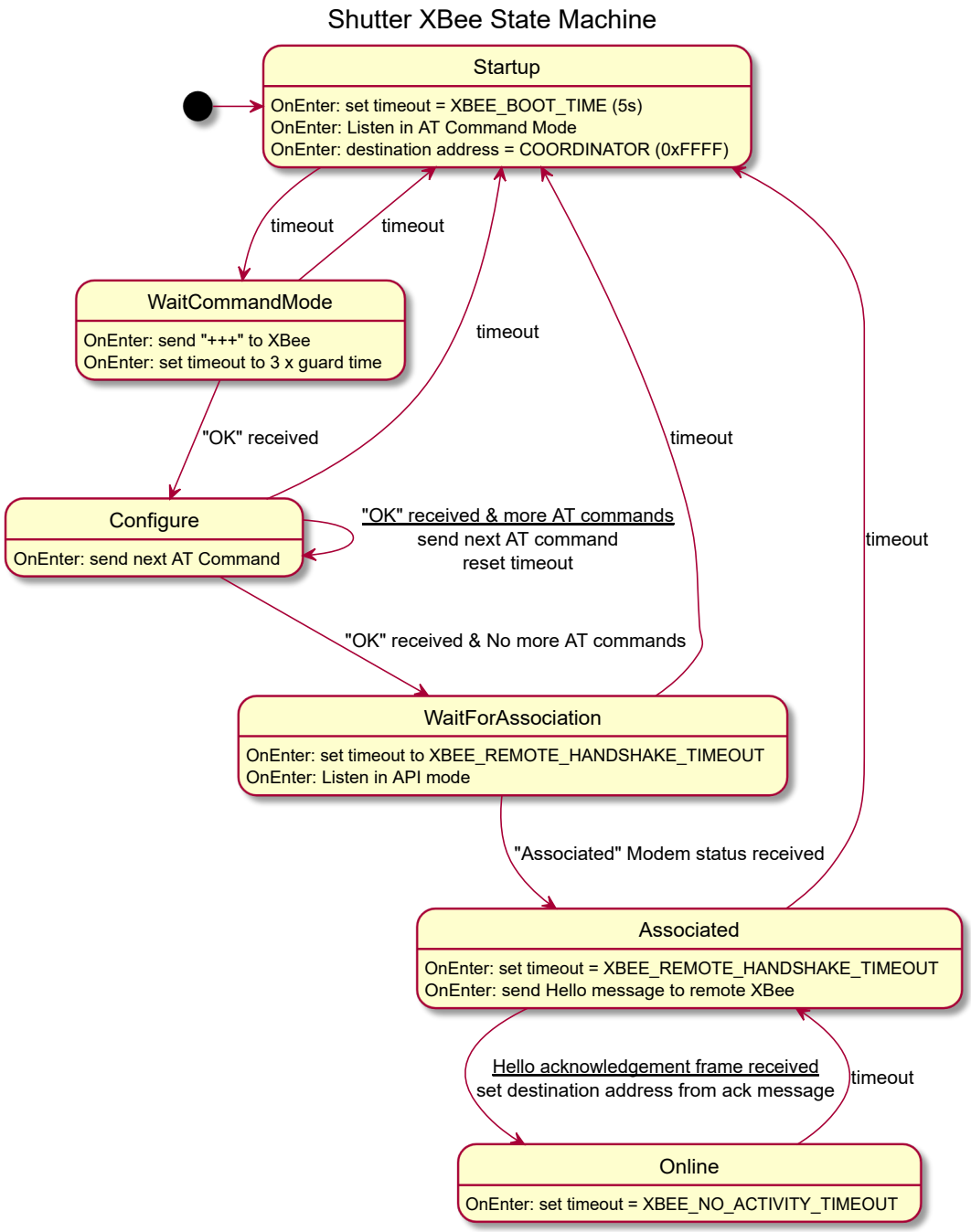
The XBee communications is managed by a state machine that controls startup, configuration and ongoing monitoring of the connection to the remote device.

The framework of the state machine is common to both rotator and shutter, so it is contained in a shared library called `XBeeStateMachine`. The operation of the two modules differs as the Rotator module must act as coordinator and the Shutter module as an endpoint. The states are therefore implemented directly in each sketch.

Rotator XBee State Machine



Shutter XBee State Machine


Shutter XBee State Machine
Startup
OnEnter: set timeout = XBEE_BOOT_TIME (5s)
OnEnter: Listen in AT Command Mode
OnEnter: destination address = COORDINATOR (0xFFFF)
timeout
timeout
WaitCommandMode
OnEnter: send "+++" to XBee
OnEnter: set timeout to 3 x guard time
timeout
"OK" received
timeout
timeout
Configure
OnEnter: send next AT Command
"OK" received & more AT commands
send next AT command
reset timeout
"OK" received & No more AT commands
WaitForAssociation
OnEnter: set timeout to XBEE_REMOTE_HANDSHAKE_TIMEOUT
OnEnter: Listen in API mode
"Associated" Modem status received
Associated
OnEnter: set timeout = XBEE_REMOTE_HANDSHAKE_TIMEOUT
OnEnter: send Hello message to remote XBee
Hello acknowledgement frame received
set destination address from ack message
timeout
Online
OnEnter: set timeout = XBEE_NO_ACTIVITY_TIMEOUT