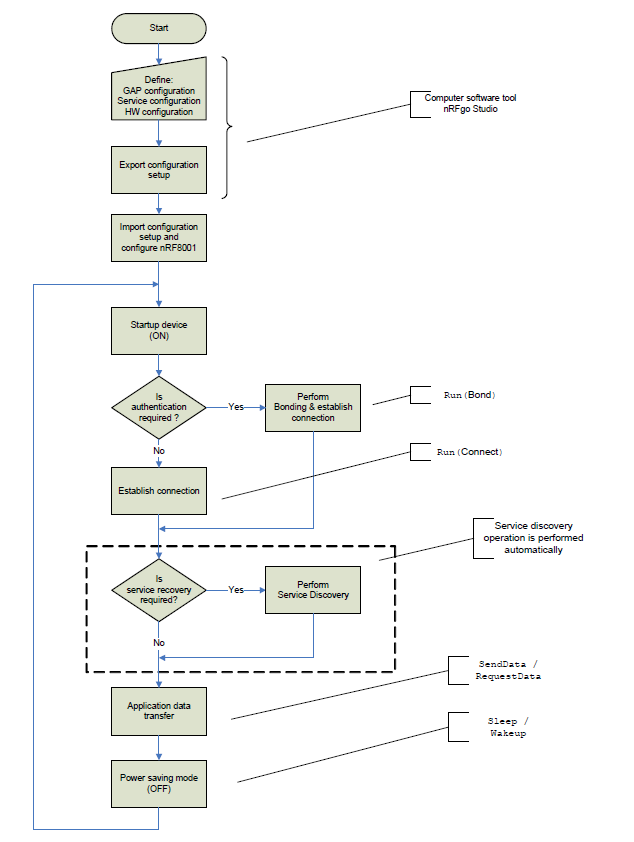
**Steps required to use the nRF8001:**

**Download the nRFgo studio from the** [**nordicsemi.com website**](http://www.nordicsemi.com/kor/Products/2.4GHz-RF/nRFgo-Studio#Downloads)**.**



**Getting started with the my\_project example in the nRF8001 SDK for Arduino**

The my\_project example, provides a skeleton project to setup the nRF8001, advertise to an iPhone and get connected to the iPhone. You can add your code to this project to send data to the iPhone. Use the functions in lib\_aci.c to use the nRF8001 command set.

**The queue interface to the nRF8001 ACI**

All ACI commands from the application to the nRF8001 are placed in the tail of the ACI command queue. All ACI events received from the nRF8001 are placed in the tail ACI Event queue. The RDYN line interrupt handler reads ACI commands from the head of the ACI command queue and and places ACI Events that are received in the tail of the ACI event queue.

When a command is placed in the command queue the REQN line to the nRF8001 is made low.

An interrupt is configured in the AVR to interrupt the cpu when the RDYN line is LOW. When the RDYN line is made low by the nRF8001, the AVR ATmega328 is interrupted, this is an interrupt on level. The ATmega328 can wakeup only on a level based interrupt and not on an edge interrupt. Other MCUs that support interrupt and wakeup from sleep on falling edge should use the falling edge in the RDYN line as the interrupt trigger.

The SPI clockout and sending of an ACI command and receiving of an ACI event with the nRF8001 is done in the RDYN line interrupt. The SPI clockout can also be done in the main context to reduce the time spend in the interrupt (see both figures below)

The ACI commands are sent using functions defined in the lib\_aci.h

The ACI events are pulled out from the ACI Event queue using the lib\_aci\_event\_get(). The application can then process the ACI Events.



Figure SPI in Interrupt context

The my\_project example implements the below steps:

1. Setup of the nRF8001. Use the utility function do\_aci\_setup() that will take the setup data generated from the nRFgo studio and setup the nRF8001. When the nRF8001 is setup successfully an ACI Device Started Event is received.
2. lib\_aci\_connect has been called to advertise to an iPhone (ACI Connect command)
   1. If you are using the ACTIVE line of the nRF8001 you can see by the activity on that line that the radio is sending beacons(advertising) and waiting for the iPhone to connect. The ACTIVE line should be configured for use in the nRFgo studio.
3. The iPhone has connected and the ACI Connected Event has been received in the lib\_aci\_event\_get()
4. The iPhone app has subscribed to the Characteristic for Notifications and the pipe is available. The ACI Event Pipe Status is received in the lib\_aci\_get\_event(). Use the lib\_aci\_is\_pipe\_available utility function to check if the specific pipe is available for use. When the pipe is available the lib\_aci\_send\_data() can be be used to send a ACI Send Data command to the nRF8001.

**Sending data from the application processor to the iPhone.**

Check that credit is available. The aci\_state.data\_credit\_variable is used to track the credits available to the application.

When aci\_state.data\_credit \_available is > 0

Use lib\_aci\_send\_data() to send data

After the nRF8001 sends the data in the ACI Send Data command to the iPhone, then iPhone radio will acknowledge the data on the air. This ack is sent as the ACI DataCredit Event to the application.

The ACI DataCredit event informs the application controller that the data has been sent successfully.

The ACI PipeError event informs that application controller that the send attempt on the data has failed.

**Receiving data from the iPhone**

The “ACI Event Data Received” is returned by the lib\_aci\_event\_get() function when the iPhone sends data to the application.

P.T.O.



Alternative architecture to run the SPI in main context. The interrupt is used to flag that the RDYN line has changed state from HIGH to LOW and the this is placed in a state variable (varRdyn). The rest of the queue based architecture is preserved. This reduces the time spent in the interrupt context as the SPI can take a few microseconds to be processed.

mcu\_sleep //mcu is sleeping

If varRdyn == 1

Run the SPI

set varRdyn = 0

m\_rdyn\_handle()



Figure Using the SPI in main context, interrupt to get state of the RDYN line (falling from HIGH to LOW)