LAST REVISED - 03/08/06 at 2:45pm

Each set of partners will implement their own version of a "bird" using the specifications listed in this document. Your compiled code will run on both you and your partner's "birds" for the Flock demonstration **during the last class, 12:30pm, on Friday, March 10 in the atrium**. Remember you need to qualify your "bird" before it can participate in the Flock demonstration. If for some reason you are unable to qualify your "bird", an alternative "bird" program will be provided so that you may receive your participation points for the demonstration.

Hints:

NOTE: Not all of the bird implementations need to be the same. The implementations must only meet the specifications outlined. Differences in bird behavior will not be penalized as long as the behavior is within the specifications contained in this document. In fact, it is encouraged that each group's implementation be slightly different as long as they meet the specifications.

Try to minimize the number of divides and mod used in your code. Calculate the needed values only once and store the results. A memory read is a lot faster than a divide and/or mod.

Do NOT use dynamic memory allocations (such as malloc) on the Atmega.

You should use enum types for default values and constants to simplify future modifications.

Implementation Details:

The Atrium Group ID will be 122. Each mote will use the same group ID, and Node ID (TOS_LOCAL_ADDRESS) will be the number on your mote's label. Assume the label is a hexadecimal number. (e.g. 11, A1, A5, 55)

Send all your packets to the TOS_BCAST_ADDR.

The first item in the data/payload section of a flock message is the address of the sender. There are two designators for the sender address in the message specifications: "Node0" and "TransmittingNodeNum". The "Node0" designator is used to signify that a packet should only be processed if it was sent from the root node (address == 0). The "TransmittingNodeNum" designator signifies that packets should be accepted from any source. The second item in all packets is the time (in milliseconds) the sending node sent the packet according to his internal clock. See Time section for more details.

In a 'SangSong' message the third item in the data/payload section is simply the sequence number of the packet that is being sent. Start the "SequenceNum" at 1 and increment each time you send this type of message.

TOS_Msg.strength is automatically updated with the packet strength during reception and is part of the TOS message struct.

For the purposes of the final project assume that Timer.start() takes milliseconds. If we give you a time of 1 second assume if you set a timer to fire in 1000ms it will fire 1 second later. For the flock to be successful we all must use the same assumptions.

The variable "`transmitPower`" in adjust globals packet should update the transmit power of your transmitter. You can set the transmission power level using the function CC1000Control.SetRFPower(int) in component CC1000ControlM.

/* Initialize the seed from the ID of the node */
async command result_t Random.init()

/* Return the next 16 bit random number */
async command uint16_t Random.rand()

When filling the message packets use the same "endian" as the Atmega.

Flock Algorithm:

Within a single "bird", the flock algorithm that you will implement is as follows:

A)	WAIT STATE (Silent while waiting) – [on reset should go to this state] Wait to receive a packet of type *AdjustGlobals* When entering WAIT STATE turn Tri-Color LED off and turn on the Red LED located on the corner of SoundBoard1. Tri-Color LED may be adjusted when in WAIT STATE by a command packet. If you receive a StopNWait packet when you are already in the WAIT STATE you should treat it as you just re-entered wait state. (Basically reset back to known state) Ignore SangSong Packets You do not need to keep track of time in this state. IF(*AdjustGlobals){* Set clock to value in AdjustGlobals packet When exiting WAIT STATE turn off the Red LED located on the corner of SoundBoard1 *Go to CLEAR STATE* *}* IF(*Command Packet*) { Perform command. Change LED and possibly sing a song. Note: all commands are processed immediately in the wait state. You do not need to schedule. Stay in WAIT STATE }
B)	CLEAR STATE clear all historical data – call the reset function on the SongDecision module clear all counter variables Set Listen timer for random amount of time (1000- 4000 milliseconds) Go to LISTEN STATE

	(NOTE: This was changed to simplify keeping state when you receive a command packet. You can now safely assume you will not get a command packet in the CLEAR STATE, and if you do receive one, just treat it as if it came in the LISTEN STATE)
C)	SING STATE Choose a song using the algorithm listed below Send the song to the Yamaha chip While singing your bird should continue to listen and collect information about what the neighbors are singing. After finished singing send a “SangSong” message. Go to LISTEN STATE IF(*Command Packet*) { Perform command. Change LED and possibly sing a song. Go to LISTEN STATE }
D)	LISTEN STATE Set listen timer for random time between [minListen, maxListen] milliseconds Listen and collect information about what the neighbors are singing When listen timer goes off go to SING STATE IF(*Command Packet*) { Perform command. Change LED and possibly sing a song. Go to LISTEN STATE }
E)	STARTLED STATE Send startled bird song to Yamaha chip. While singing your bird should continue to listen and collect information about what the neighbors are singing. ~~Turn Tri-Color LED off~~ Maintain the same color of the Tri-Color LED and turn on the Red LED located on the corner of SoundBoard1. Send Startled packet (remember to decrement HopCount) when finished singing the startled song. After finished singing, delay 10 seconds, turn off Red LED in the corner of SoundBoard1. Go to LISTEN STATE

There are 5 types of Active Messages your program must handle; they are as follows:

AM #	Flock Message / Packet
42	SangSong - The "I sang song" message; a message from some other birdie indicating what song it sang. You also send this packet after singing. **Notes:** **1) If you are sending this after a packet Command Packet (#52), set all data = 0 except, transmittingNodeNum, currentTime, sequenceNum, songNum.** **2) The contents of a SangSong packet should be calculated when the song decision is made right before singing, NOT when the packet is sent. Make sure to get the information from the SongDecision module.** **3) The SongDecision module does NOT populate the transmittingNodeNum, currentTime, or sequenceNum as those variables should be kept in the control module. The songNum and songPointValue variables (what your bird is actually singing) is populated when you call either the minPointSong() or the maxPointSong() function. NOTE: When populating the module assumes that you follow the choice it returned.** `uint16_t transmittingNodeNum local # of node singing` `uint32_t currentTime`

	`uint16_t  sequenceNum  start at 1, increment each time you send this packet` `uint16_t  songNum  song# that was sung` `uint16_t  songPointValue  usually same as maxPoint` `uint16_t  maxPointSongNum  Song with highest point value` `uint16_t  maxPointValue` `uint16_t  runnerUpSongNum  Runner-up song with second highest` `uint16_t  runnerUpValue` `uint16_t  minPointSongNum  Song with the lowest point value` `uint16_t  minPointValue`
50	AdjustGlobals - A message from Node 0 containing global parameters for all birds. **This message should be processed in any state.** `uint16_t  Node0` `uint32_t  currentTime` `uint16_t  Repetition  default 3` `uint16_t  minListen  default 2000 millisec.` `uint16_t  maxListen  default 15000 millisec.` `uint16_t  Threshold  default 600` `uint16_t  minThreshold  default 100` `uint16_t  Probability  default 10` `uint16_t  Silence  default 10` `uint16_t  transmitPower  default 0x0F` `uint16_t  startledHopCount  default 1`
51	StopNWait - A message from Node 0 telling you to stop and go to the WAIT STATE. **This message should be processed in any state.** `uint16_t  Node0  On receipt, go to WAIT STATE` `uint16_t  startAddr  Beginning of Address Range` `uint16_t  endAddr  End of Address Range`
52	Command Packet - A message from Node 0 telling you to adjust your light and possibly play a song. **NOTES:** **1) This message should be processed in any state.** **2) When finished processing the command the bird should remain in the same state it was in before the command packet was received (e.g. WAIT or LISTEN).** **3) Refer to Command Packet description below** `uint16_t  Node0` `uint32_t  currentTime` `uint16_t  startAddr  Beginning of Address Range` `uint16_t  endAddr  End of Address Range` `uint16_t  LED  LED state` `uint16_t  song  Song to Play` `uint32_t  playtime  Song should start playing at`
60	Startled - a message from some other birdie indicating that they have been startled. Stop what you are doing and sing your startled message. After the startled message you should go to the LISTEN_STATE. Do NOT send a SangSong packet for the startle song. Instead send the startled message. Remember to decrement the HopCount when you send the message. On reception, If (HopCount > 0) process message Else ignore message

	uint16_t transmittingNodeNum local # of startled node uint32_t currentTime uint16_t hopCount — Number of hops remaining. If startled by radio message. Decrement value before sending. uint16_t startledSeqNum — A randomly generated number that is stored to check to see if you have been startled by this startle packet before
61	Identification – After the light sensor has detected the identification sequence it sends this packet to let the control software know the nodeID that has been identified. – Your node should ignore any packet of type 61. uint16_t transmittingNodeNum local # of indentified node

Song Decision Algorithm:

The course staff has provided a "SongDecision" module that keeps statistical data on the flock to identify which bird song is being played the most and least. The module also utilizes a do-not sing list to ensure the flock won't be stuck singing the same song. The overall flock uses random numbers to produce an emergent behavior. You will need to implement the following song selection algorithm:

```
x = rand() % Probability
y = rand() % Silence

if (x == 0)
      SONG = song with the lowest point value
else if (y == 0)
      Silence… Don't sing a song-- go back to LISTEN STATE.
else
      SONG = song with the highest point value
```

Description of Command Packet:

The command packet is used by the command software to instruct a single or multiple mote(s) to perform a specific action. To determine if the message was intended for your node check if the node ID falls within the instruction's address range. (i.e. startAddr <= nodeID <= endAddr). If your node does fall within the command address range you must process the LED and sing instruction.

The LED variable contains the color value for the tri-color LED:

bit 0 controls the Blue LED in the TriColor LED
bit 1 controls the Green LED in the TriColor LED
bit 2 controls the Red LED in the TriColor LED

The command packet may also include an instruction to play a song. The song field will either contain the song number to play or 0xffff if no song should be played. If a song is

requested [bird songs(0-15), startle (16), special(17)], check the playTime to determine when the song should be played. If the time is less than the mote's internal time play then play song immediately. If the time is greater than the mote's internal time then schedule the song to begin at the time specified in playTime. Assume only one sing event can be scheduled at a time. Note: in the wait state (where no time is kept) you can execute all commands immediately with no need to schedule.

After singing a song triggered by a command packet, your bird should send a SangSong message with only the first four fields: `transmittingNodeNum`, `currentTime`, `sequenceNum`, `songNum`. The remaining fields should be filled with zeros. You may send the SangSong message immediately after you finish singing the song. When triggered by a command packet you should send a SangSong message for all songs 0-17, do not send a startled message for the startled song. The SangSong message being sent after a command packet should only populate the first four fields of the packet. The rest of fields in the packet should be 'zero's.

Method to Synchronize Time:

Each node should keep track of time by incrementing a uint32_t every millisecond.

We can create a loose synchronization scheme by each node keeping its own internal time that is continually updated by its neighbors towards an overall network wide consensus time. When a new message arrives from another device with a different global time each device assumes they are only partially right. The devices then average their local time with the other device's time to produce a revised local time. Note that individual devices do not need to be aware of the actual world time as long as they all use identical and uniform time divisions and can agree on a global consensus time.

When the mote first boots it should not start its internal clock at zero. Instead, it should wait to receive the time from an adjustGlobals packet to avoid problems with the network attempting to average a node that may be radically different.

We also want to avoid a node that maybe malfunctioning from drastically changing the network's time synchronization, so your code should ignore any time value that is more than 3000 milliseconds different than the mote's internal time value.

We recommend using an interrupt from Timer1 to increment your time variable as the timer module in TinyOS is not accurate enough for a 1ms scale since it is implemented with tasks.

Method to Identify Node:

To easily identify deployed nodes your program should include two methods of identification: 1) a packet from the control software should cause the mote to identify itself and 2) the mote should identify itself to the control software. The first method (control software to mote) is accomplished by the control software sending a command packet with the same 'start address' and 'end address' causing the specified mote to move to a uniquely identifiable state. The second method requires the mote to detect that an identification packet should be sent to the control software. The mote should sense that it

is being identified by detecting large light changes that form a pattern. A user will cover and uncover the light sensor 4 times in a row to signal that an identification packet should be sent to the control software. This means that 8 transition events need to be detected in order. After detecting the sequence you should turn on the mica2dot's red LED for 5 seconds to give feedback to the user. A sampling rate of 3 times a second should be fast enough to detect a person covering and uncovering the light sensor. To make sure random shadows do not cause the mote to falsely identify itself a transition event is defined by the 4 most significant bits of the light level changing by more than 4. In addition, your program should also have a timeout period of 2 seconds between transition events to make sure the events being detected are part of a continuous sequence. After a timeout your state machine should reset. Notice that the light sensor is used to both identify the bird and startle the bird; this means that you will startle the bird when trying to perform the identification sequence when not in the WAIT_STATE. To avoid the entire flock startling when someone is trying to identify a bird, do not send the startle packet until after your algorithm has determined it's not an identification packet. Unlike startle you should be able to identify the node in any state. NOTE: If for some reason you program misses and edge with a timeout of 2 seconds, it will most likely pick up the next edge change. Therefore no points will be deducted if it takes 5-6 times to identify the mote as long as the user interface is reasonable.

Description of how the "SongDecision" Module works:

The course staff has provided a "SongDecision" module that will help your program pick the next song by calculating the 'strongest' and 'weakest' songs in the flock through a point scoring system. Points are assigned based on what the surrounding nodes are singing. The module also provides statistical feedback on your nodes point values.

When not in the WAIT STATE, your program should always be listening on the radio to the surrounding nodes to collect information about what songs the nodes are playing. From the SangSong packet (AM #42 type packets) you should pass the following information to the SongDecision Module:

```
uint16_t    TransmittingNodeNum
uint16_t    songNum
uint16_t    TOS_msg_strength
```

The SongDecisions module uses a 64-entry circular FIFO queue, and each new entry writes over the oldest entry in the queue.

To determine the next song to sing the SongDecision module determines a point value based on the number of times the song has been sang and how close in proximity the node was that sang the song(via signal strength). Your program should call the functions maxPointSong() to get the song that has the highest point value and the function minPointSong() to get the song with the lowest point value.

The SongDecision module utilizes the variables from the adjustGlobals packet to make sure the flock algorithm is not dominated by a single song. This means the SongDecision

module needs to keep track of the songs that have been popular so that it can force the flock to move to another song. The “Threshold” and “Repetition” values from the adjust globals packet are meant to ensure that songs will be allowed to propagate through the flock, but then eventually die off. This growth and die off is accomplished by limiting the number of song repetitions once a song’s point value crosses the “Threshold”. The “Repetition” count limits the number of times a song can play; thereby, allowing a strong song to propagate to a large number of nodes and then die off once a song becomes repetitive. The “minThreshold” parameter allows the flock to be adjusted by establishing a minimum number of points that a song must to be counted as “chosen”.

To accomplish the divergence the system keeps a FIFO list of 3 songs that will basically act as a do-not sing list once the song has been selected too many times (aka “Repetition” times). A song is added to the list when it has been selected with a point score greater than the threshold. The program will keep a count of the number of times the song has been chosen with a point value over the “Threshold” value. Once the song has been chosen “Repetition” times the system will pick a new song. The function in the SongDecision module will automatically return a random song that is not on the do-not sing list if the original song with max point value is on the do-not sing list. Once a new song has a point value above the Threshold repetitions times it will push the oldest song off the do-not-sing list. You will always be able to sing something, because the system only tracks the last three songs sung greater than Repetition times, which means there are thirteen songs not on that list that it can randomly choose from. The nice thing is that the do-not sing list causes the strongest songs to die off. NOTE: Both the FIFO and do-not sing list are cleared when you call the reset function.

To select a song, first issue an “updateSongSelections” command to make the “SongDecision” module recalculate the song statistics. Then wait until the “SongDecision” module indicates it has finished by signaling an “updateSongSelectionsComplete” event. Next choose a song by calling either the “minPointSong” or the “maxPointSong”. Finally, obtain the song statistics from the “SongDecision” module by calling “getSongStatistic”. NOTE: The order the function are called is important. Do NOT call “minPointSong” or “maxPointSong” before “updateSongSelectionsComplete” has been signaled. Also, do NOT call “getSongStatistic” before “minPointSong” or “maxPointSong.” It is fine to call another “updateSongSelections” without choosing a song as long as the “updateSongSelectionsComplete” event has already been signaled from the previous call.

Method for determining when to startle a bird:

A startle should only occur in the LISTEN or SING STATES. There are two methods to startle a bird: 1) Light Sensor 2) Radio

Use the ADC to trigger the bird being startled by detecting when there has been a change in the light level. A startled should be triggered when the 4 most significant bits of the ADC value changes by 3 or more. You should be sampling the ADC approximately 3 times a second. If a startle occurs go immediately to STARTLED_STATE and sing the

startled song. After you finish singing the startled song then send a startled packet and return to the normal sing and listen. You do not send a SangSong packet for a startled, only send a startle packet. To avoid the entire flock startling when someone is trying to identify a bird, do not send the startle packet until after your algorithm is sure that the light change is not associated with a startle. In other words, play the startle song (unless in wait) and hold off on the transmission of the startle packet until the timeout lets you know it's not an identification. NOTE: Only the initiator of the startled sequence (ie. Node which detected a change in light level) should set the StartledSequenceNum.

The startle detection mechanism should only detect one startle for a single movement. We want to avoid two startles being caused by the same movement. For example if someone puts their hand over the bird an edge transition will be detected as the light level is reduced (the bird becomes startled) and as the person takes his hand away the light level increases causing a second edge detection (second startle event). A person walking by a bird obscuring the light should also only cause one startle song to be played. You will need to come up with a solution to this problem so the bird is only startled once. You will need to keep the startle mechanism working the same so that the bird can still be startled by quickly increasing or decreasing the light level. Reasonableness should be used when designing your solution. If someone leaves their hand over the bird for 1 hour or even 15 seconds it is fine to count the hand decreasing the light level as one startle and the hand moving away as a second startle. The goal is to design a solution so that a reasonable movement will only be detected once.

If you bird receives the startled packet then your bird should check to see if 1) (HopCount > 0) and 2) (StartledSeqNum != previousStartledSeqNum). Basically check if your bird has already been startled by this startled sequence so that a birdie is not continually startled by the same initial startle. If these two checks are true, then your bird should become startled. After you finish singing the startled song, decrement the HopCount, store the startledSeqNum, and send a startled packet if HopCount is > 0, then return to the normal sing and listen.

Bird Calls/Songs:

All of the song information that you will need to use in this project is in the birdsongs.h file. There are three arrays of note: a short array containing the startle song (this can be kept in static memory), a large array containing a special song and a 2-dimensional array of the regular 16 songs you have been using for the past several labs. Note: The special song is only to be played when indicated by a command message and should not be a part of the regular song rotation. The 2 large arrays should be kept in flash memory as indicated by their array declarations.

Also included in this file are the register initializations that must be included in your code to make the startle and special songs play. These simply reset the tempo and volume registers and will need to be executed prior to loading any notes for their corresponding

song in the FIFO. In the case of the startle song, you have been given a function that will adjust the proper registers and load the entire song into the FIFO since the entire startle song fits into the FIFO (you may use this function if you wish). In the case of the special song, you have only been given a function that adjusts the register values.

A second file has been included: timbres.c. This file contains the initialization code for the timbre and timbre-allotment registers (it also includes tempo and volume register default settings). This code will replace the initialization of timbres, timbre allotment, etc. in HPLSoundBoardM.nc.

The more astute among you will notice that the initialization above sets 5 different timbres. However, as you should all know the Yamaha chip is only capable of using 4 timbres. Well that is almost true, only 4 timbres can be allotted at a time, but the timbre allotment can be changed in the middle of playing a song. If you look at the first rest value of each song in the birdsongs.h file, you will find that its value is designed to make sure the correct timbre allotment is used for each file.