# The SPINE Manual Version 1.2

*The SPINE Team*

September 2008

1 Introduction ........ 3
1.1 How to install SPINE ........ 4
1.1.1 Node Side ........ 5
1.1.2 Server Side ........ 5
2 How to use SPINE ........ 6
2.1 How to run a simple application using SPINE1.2 ........ 7
3 SPINE architecture ........ 11
3.1 Communication ........ 12
3.2 Sensing ........ 16
3.2.1 How to introduce a new sensor ........ 16
3.3 Processing ........ 19
3.3.1 Feature Engine ........ 20
3.3.2 Alarm Engine ........ 23
3.3.3 How to add a new function ........ 25

# 1 Introduction

SPINE (*Signal Processing In Node Environment*) is a framework for the distributed implementation of signal processing algorithms in wireless sensor networks.
It provides a set of on node services that can be tuned and activated by the user depending on application needs.
SPINE is released as Open Source project under LGPL 1.2 license and is available on line at http://spine.tilab.com.

If you are willing to contribute to the Open Source project, please email spine-contrib@avalon.tilab.com by specifying a short work plan with a description of the new functionalities, the impact on existing code, and the expected time for the first stable release.

The SPINE framework has two main components

1. Sensor Node side. It is developed in TinyOS2.x environment and provides on node services such as sensor data sampling and storage, data processing and more;

2. Server side. It is developed in Java SE and acts as coordinator of the sensor networks. Therefore, it manages the network, set up and activate on node services depending on the application requirements and more.

The framework has been redesigned and the newest release (1.2) provides many more levels of expansibility than before.

The core framework is now organized into three main parts that take care of different aspects, namely the communication, the sensing and the processing parts.
The new structure reflects into the file organization, especially on the sensor node side (
Figure 1.1).

```
Spine_nodes
|__apps
|     |__SPINEApp
|__support
|     |__make
|__tos
|     |__interfaces
|     |     |__communication
|     |     |__processing
|     |     |__sensing
|     |     |__utils
|     |__platforms
|     |__sensorboards
|     |__system
|     |     |__communication
|     |     |__processing
|     |     |__sensing
|     |     |__utils
|     |__types
```

**Figure 1.1 : Spine_nodes1_2 source code organization**

The Server Side has a slightly different organization (Figure 1.2).
This structure reflects the need of having the framework logic not depending on the kind of network it is communicating with. In other words, the core implementation of SPINE does not use any TinyOS specific APIs and can be run independently on the underlying protocol stack (e.g. ZigBee networks).
Platform-independent code may be found into:

- spine package contains SPINE core logic
- `spine.datamodel` package contains data entities used by the framework
- `spine.datamodel.functions` sub-package defines the structure of the function

At this stage, SPINE1.2 server side provides an implementation for TinyOS2.x network; therefore it provides the support for TinyOS low level communication:

- `spine.communication.tinyos` contains TinyOS specific logic (packet format and parse/build operations) and low level communication procedures (calling tinyos.jar APIs).

```
Spine_serverApp
|
|__src
|    |
|    |__spine
|    |      |
|    |      |__communication.tinyos
|    |      |
|    |      |__datamodel
|    |      |
|    |      |__datamodel.functions
|    |
|    |__test
|
|__lib
|
|__jar
|
|__doc
|
|__defaults.properties
|
|__build.xml
|
|__build.properties
```

**Figure 1.2: Spine_serverApp1_2 code organization**

SPINE1.2 release provides also the SPINE.jar that can be imported in any project that uses SPINE APIs and the full javadoc documentation.

## 1.1 How to install SPINE

1. Download SPINE 1.2 from SPINE website (http://spine.tilab.com/).
2. The unzipped spine folder contains:
    a. Spine_nodes folder with TinyOS2.x code to be run on the motes
    b. Spine_serverApp folder with Java code to be run on a computer

c. COPYING and License text files containing info about the licensing
d. SPINE_manual

### 1.1.1 Node Side

Spine_nodes contains code to be compiled in TinyOS2.x and then flashed on sensor nodes. Spine_nodes 1.2 has been developed and tested with TinyOS version 2.1.0. Older TinyOS2.x versions have also been tested, and Makefile can be configured to support an older version, but the SPINE Team strongly suggests to use to TinyOS2.1.0 release.

1. Copy Spine_nodes folder into your `tinyos-2.x-contrib` folder
2. From the app/SPINEApp folder compile and install SPINE1.2 framework on your platform. At this time, platforms supported by SPINE1.2 are
    a. Telosb motes with spine sensor board
       `SENSORBOARD=spine make telosb`
    b. Telosb motes with biosensor sensor board
       `SENSORBOARD=biosensor make telosb`
    c. Micaz motes with mts300 board
       `SENSORBOARD=mts300 make micaz`
    d. shimmer motes
       `SENSORBOARD=shimmer make shimmer`

If you want your specific sensor or mote platform to be supported by SPINE, please refer to 3.2.1 for further details and email spine-dev@avalon.tilab.com if you have specific questions.

### 1.1.2 Server Side

Spine_serverApp contains the Java code for running the server side (e.g. coordinator) of a SPINE network.

1. `src` contains SPINE1.2 source code organized into
    a. spine
    b. test
2. `defaults.properties` contains the framework properties
3. `lib`: contains a jar file that SPINE must include
4. docs: contains SPINE1.2 javadoc documentation
5. `jar`: contains the framework jar file
6. `build.properties` and build files for ant

You can compile and run SPINE framework and its test application either using ant textual ant commands or creating a java project using a IDE (such as Eclipse, NetBeans…).

Please note that you have to **add an external jar** (`tinyos.jar`) to your project. This jar is not part of the SPINE distribution and can be found in the tinyos2.x\support\sdk\java folder or downloadable at http://tinyos.cvs.sourceforge.net/tinyos/tinyos-2.x/support/sdk/java/.

For any problems and specific questions about the installation, please send e-mail to spine-dev@avalon.tilab.com

# 2 How to use SPINE

SPINE framework provides, on the Server Side, simple Java APIs to develop applications on the coordinator. Therefore, the main strength of the SPINE framework is to allow users to be ready to develop applications in sensor networks without bothering with node-side programming. Of course, SPINE Node Side source code is distributed as well and next paragraph will give details about it.
Developers can easily form, manage and collect data from the sensors in the network writing a simple Java program: no more firmware programming is needed!
On the Java side, the user can develop its own application that will have to implement the SPINEListener interface and can use any of the API provided by the SPINEManager.

Since the application on the server side must implement the SPINEListener interface, it has to implement the following methods:

| **Method Summary** | |
|---|---|
| `void` | **`dataReceived`**`(int nodeID, Data data)` This method is invoked by the SPINEManager to its registered listeners when it receives new data from the specified node. |
| `void` | **`discoveryCompleted`**`(java.util.Vector activeNodes)` This method is invoked by the SPINEManager to its registered listeners when the discovery procedure timer fires. |
| `void` | **`newNodeDiscovered`**`(Node newNode)` This method is invoked by the SPINEManager to its registered listeners when it receives a ServiceAdvertisement message from a BSN node |
| `void` | **`serviceMessageReceived`**`(int nodeID, ServiceMessage msg)` This method is invoked by the SPINEManager to its registered listeners when a ServiceMessage is received from a particular node. |

Then, the application can use any APIs exposed by the SPINEManager:

| **Method Summary** | |
|---|---|
| `void` | **`activateFunction`**`(int nodeID, SpineFunctionReq functionReq)` Activates a function (or even only function sub-routines) on the given sensor. |
| `void` | **`bootUpWsn`**`()` Currently, it does nothing! |
| `void` | **`deactivateFunction`**`(int nodeID, SpineFunctionReq functionReq)` Deactivates a function (or even only function sub-routines) on the given sensor. |
| `void` | **`deregisterListener`**`(SPINEListener listener)` Deregisters a SPINEListener to the manager instance |
| `void` | **`discoveryWsn`**`()` Commands the SPINEManager to discovery the surrounding WSN nodes |
| `java.util.Vector` | **`getActiveNodes`**`()` Returns the list of the discovered nodes as a Vector of spine.datamodel.Node objects |
| `static SPINEManager` | **`getInstance`**`(java.lang.String[] args)` Returns the SPINEManager instance connected to the given base-station Those parameters should be retrieved using the Properties instance obtained thru the static SPINEManager.getProperties method |
| `static Properties` | **`getProperties`**`()` Returns an instance of a Properties implementation class which can be queried for retrieving system and framework properties and parameters |

| | |
|---|---|
| void | **readNow**(int nodeID, byte sensorCode)<br>Commands the given node to do a 'immediate one-shot' sampling on the given sensor. |
| void | **registerListener**(SPINEListener listener)<br>Registers a SPINEListener to the manager instance |
| void | **resetWsn**()<br>Commands a software reset of the whole WSN. |
| void | **setDiscoveryProcedureTimeout**(long discoveryTimeout)<br>This method sets the timeout for the discovery procedure. |
| void | **setupFunction**(int nodeID, SpineSetupFunction setupFunction)<br>Setups a specific function of the given node. |
| void | **setupSensor**(int nodeID, SpineSetupSensor setupSensor)<br>Setups a specific sensor of the given node. |
| void | **start**(boolean radioAlwaysOn)<br>Starts the WSN sensing and computing the previously requested functions. |
| void | **start**(boolean radioAlwaysOn, boolean enableTDMA)<br>Starts the WSN sensing and computing the previously requested functions. |
| boolean | **started**()<br>Returns true if the manager has been asked to start the processing in the wsn; false otherwise |
| void | **synchrWsn**()<br>Commands a software 'on node local clock' synchronization of the whole WSN. |

Examples about which function can be set, which data can be received and other details can be found in the SPINETest application which comes with the release and is illustrated in the following chapter. More examples about how to use the Java side are given all through this document.

For further details about the Java side, please refer to the Javadoc documentation that can be found in the release.

## 2.1 How to run a simple application using SPINE1.2

The SPINE1.2 release comes with a simple test application that can be easily run to experiment the framework basic functionalities.
Take the following steps:

1. compile and flash on your platform the SPINE1.2 node side framework;
2. compile and flash a TinyOS2.x BaseStation. Please check that sensor nodes and base station are both working on the same radio channel and the same TinyOS version has been used for flashing all the nodes.
3. plug the BaseStation to your computer and type “motelist” from your shell: this will tell you your port number;
4. open the `Spine_serverSide/defaults.properties` file and set the MOTECOM parameter and LocalNodeAdapter class name according to one of the following options depending if you are using the serial forwarder on a Linux or Windows machine (a) or directly communicating with the serial port on your PC using a Windows machine (b)
   a. `MOTECOM=sf@127.0.0.1:9002`
      `LocalNodeAdapter_ClassName=spine.communication.tinyos.SFLocalNodeAdapter`
   b. `MOTECOM=serial@COM41:telosb`
      `LocalNodeAdapter_ClassName=spine.communication.tinyos.TOSLocalNodeAdapter`

   Option b may be used also on a Linux machine, but you need to build libgetenv and libtoscomm from your tinyos before you can install and run any SPINE application.

   ```
   cd $TOSROOT/support/sdk/java && make
   sudo tos-install-jni
   ```

5. edit Spine_serverSide/test/SPINETest.java and go through the code if you want to customize the test application. The code documentation helps to understand what functionalities SPINE exposes to the java developer.

As mentioned before, SPINETest.java implements the SPINEListener interface and uses SPINEManager APIs for managing and communicating with the nodes in the network.

The SPINETest provided within the SPINE 1.2 release performs the following actions:

a. a discovery message is broadcasted to check how the PAN is composed:

```
manager.discoveryWsn();
```

b. when the discovery is completed, all the received info about nodes present in the PAN is displayed.

```
curr = (Node)activeNodes.elementAt(j);
// we print for each node its details (nodeID, sensors and functions
provided)
System.out.println(curr);
```

The information displayed at this point is:

i. node id
ii. supported sensors
iii. supported functionalities

c. if a node with an accelerometer is found:

i. the accelerometer is set with sampling time SAMPLING_TIME=50 msec

```
SpineSetupSensor sss = new SpineSetupSensor();
sss.setSensor(sensor);
sss.setTimeScale(SPINESensorConstants.MILLISEC);
sss.setSamplingTime(SAMPLING_TIME);
manager.setupSensor(curr.getNodeID(), sss);
```

ii. the feature engine function is set on that node to work on data coming from the accelerometer sensor with window WINDOW_SIZE=40 and shift SHIFT_SIZE=20

```
FeatureSpineSetupFunction ssf = new FeatureSpineSetupFunction();
ssf.setSensor(sensor);
ssf.setWindowSize(WINDOW_SIZE);
ssf.setShiftSize(SHIFT_SIZE);
manager.setupFunction(curr.getNodeID(), ssf);
```

iii. few features are activated on that node on the accelerometer data (MODE, MEDIAN, MAX and MIN on all the accelerometer's channels)

```
FeatureSpineFunctionReq sfr = new FeatureSpineFunctionReq();
sfr.setSensor(sensor);
sfr.addFeature(SPINEFunctionConstants.MODE,
   ((Sensor)curr.getSensorsList().elementAt(i)).getChannelBitmask());
sfr.addFeature(SPINEFunctionConstants.MEDIAN,
   ((Sensor) curr.getSensorsList().elementAt(i)).getChannelBitmask());
sfr.addFeature(SPINEFunctionConstants.MAX,
   ((Sensor) curr.getSensorsList().elementAt(i)).getChannelBitmask());
sfr.addFeature(SPINEFunctionConstants.MIN,
   ((Sensor) curr.getSensorsList().elementAt(i)).getChannelBitmask());
manager.activateFunction(curr.getNodeID(), sfr);
```

iv. more features are activated (MEAN,AMPLITUDE)

```
FeatureSpineFunctionReq sfr = new FeatureSpineFunctionReq();
sfr.setSensor(sensor);
sfr.addFeature(SPINEFunctionConstants.MEAN,
      ((Sensor) curr.getSensorsList().elementAt(i)).getChannelBitmask());
sfr.addFeature(SPINEFunctionConstants.AMPLITUDE,
      ((Sensor) curr.getSensorsList().elementAt(i)).getChannelBitmask());
manager.activateFunction(curr.getNodeID(), sfr);
```

v. the alarm engine function is set on the node to work on data coming from the accelerometer sensor with window WINDOW_SIZE=40 and shift SHIFT_SIZE=20. Please note that Feature and Alarm engines can be set with different settings, since they are two separate components. However, in this test application, they have been set with the same value to better check the results.

```
AlarmSpineSetupFunction ssf2 = new AlarmSpineSetupFunction();
ssf2.setSensor(sensor);
ssf2.setWindowSize(WINDOW_SIZE);
ssf2.setShiftSize(SHIFT_SIZE);
manager.setupFunction(curr.getNodeID(), ssf2);
```

vi. two alarms are set on the accelerometer sensor, so that an alarm message will be sent back when:
   1. the MAX value on CH1 is greater than upperThreshold value = 40
   2.

```
AlarmSpineFunctionReq sfr2 = new AlarmSpineFunctionReq();
sfr2.setDataType(SPINEFunctionConstants.MAX);
sfr2.setSensor(SPINESensorConstants.ACC_SENSOR);
sfr2.setValueType((SPINESensorConstants.CH1_ONLY));
sfr2.setLowerThreshold(lowerThreshold);
sfr2.setUpperThreshold(upperThreshold);
sfr2.setAlarmType(SPINEFunctionConstants.ABOVE_THRESHOLD);
manager.activateFunction(curr.getNodeID(), sfr2);
```

   3. the AMPLITUDE on CH2 is lower than lowerThreshold value = 2000

```
sfr2.setDataType(SPINEFunctionConstants.AMPLITUDE);
sfr2.setSensor(SPINESensorConstants.ACC_SENSOR);
sfr2.setValueType((SPINESensorConstants.CH2_ONLY));
sfr2.setLowerThreshold(lowerThreshold);
sfr2.setUpperThreshold(upperThreshold);
sfr2.setAlarmType(SPINEFunctionConstants.BELOW_THRESHOLD);
manager.activateFunction(curr.getNodeID(), sfr2);
```

d. if a node with internal CPU temperature sensor is found:
   i. the temperature sensor is set with sampling time OTHER_SAMPLING_TIME =100 msec
   ii. the feature engine function is set on that node to work on data coming form the temperature sensor with window OTHER_WINDOW_SIZE=80 and shift OTHER_SHIFT_SIZE=40
   iii. few features are activated on that node on the temperature data (MODE, MEDIAN, MAX and MIN)
   iv. the alarm engine function is set on the node to work on data coming from the accelerometer sensor with window WINDOW_SIZE=40 and shift SHIFT_SIZE=20.

v. Then one alarm is set on the internal CPU temperature sensor, so that an alarm message will be sent back when:
   1. the MIN value on CH1 is greater than 1000 and lower than 3000

e. once all the request are set, the network starts

```
manager.start(true, true);
```

f. on reception of the activated data (`dataReceived`), data payload is displayed

```
System.out.println(data);
```

g. during application runtime, functions can be deactivated and activated. Here for instance:
   i. After receiving 5 feature packets, the first activated feature on that sensor is deactivated

```
   if(counter == 5) {
   // it's possible to deactivate functions computation at runtime (even when
the radio on the node works in low-power mode)
   FeatureSpineFunctionReq sfr = new FeatureSpineFunctionReq();
   sfr.setSensor(features[0].getSensorCode());
   sfr.removeFeature(features[0].getFeatureCode(),SPINESensorConstants.ALL);
   manager.deactivateFunction(nodeID, sfr);
   }
```

   ii. After receiving 10 feature packet a new feature (RANGE) is computed on the first channel of that sensor

```
   if(counter == 10) {
   // and, of course, we can activate new functions at runtime
   FeatureSpineFunctionReq sfr = new FeatureSpineFunctionReq();
   sfr.setSensor(features[0].getSensorCode());
  sfr.addFeature(SPINEFunctionConstants.RANGE,SPINESensorConstants.CH1_ONLY);
   manager.activateFunction(nodeID, sfr);
   }
```

   iii. After 20 alarm packets the

```
   if(counter_alarm == 20) {
   AlarmSpineFunctionReq sfr2 = new AlarmSpineFunctionReq();
   sfr2.setSensor(SPINESensorConstants.ACC_SENSOR);
   sfr2.setAlarmType(SPINEFunctionConstants.ABOVE_THRESHOLD);
   sfr2.setDataType(SPINEFunctionConstants.MAX);
   sfr2.setValueType((SPINESensorConstants.CH1_ONLY));
   manager.deactivateFunction(nodeID, sfr2);
      }
```

# 3 SPINE architecture

SPINE is a framework for signal processing in wireless sensor networks; therefore it has to take care of communication among nodes into the network, on-node sensor management and signal processing functionalities.
SPINE1.2 node side architecture reflects all these functionalities and divides on-node functionalities into three main logic blocks:

1. Communication: takes care of radio communication, data packet build and parse, radio duty cycle and channel access schemes;
2. Sensing: manages data sampling from sensors and storage on a shared buffer;
3. Processing: takes care of the data processing

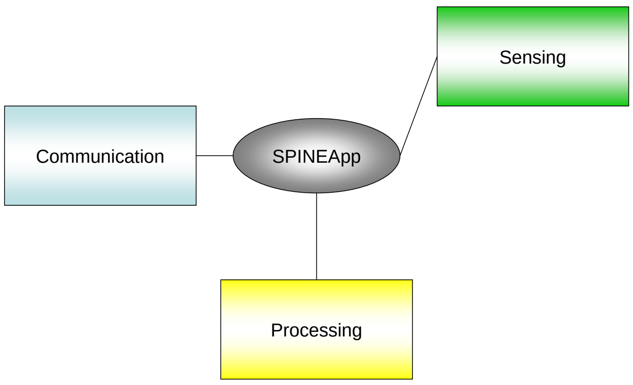

**Figure 3.1: SPINE 1.2 node side, high level architecture**

The lifecycle of SPINE1.2 framework on the node side is managed by the SPINEApp component.
SPINEApp acts as a dispatcher among the aforementioned modules and in particular it provides the following services:

- Upon receipt of the Service Discovery message, asks for information about supported sensors to the SensorBoard Controller and about supported functionalities to the Function Manager and then sends back to the coordinator the Service Advertisement packet.
- Dispatch incoming packet depending on their types:
    - SetUpSensor messages to the Sensing modules (Sensor Board Controller);
    - SetUpFunction and ActivateFunction messages to the Processing modules (Function Manager)
- Handle start and reset message (sync message still need to be supported)

SPINEApp code may be found into the `Spine_nodes/apps` folder.
This is the place form where you'll type the make and make install commands.

## 3.1 Communication

On the node side, SPINE 1.2 communication part (Figure 3.2) is composed of two main components: the Radio Controller and the Packet Manager.

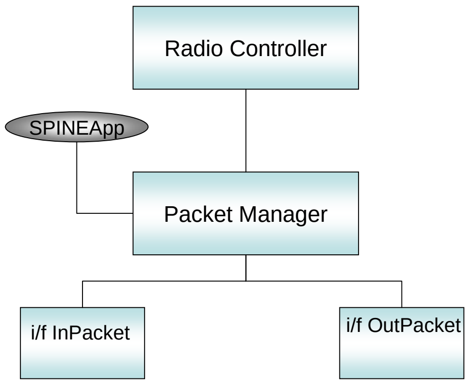

**Figure 3.2: SPINE 1.2 node side, Communication Modules**

The **Radio Controller** takes care of the access to the radio (send and receive operation) and, when activated, the low power mode and the TDMA access to the radio.

Upon receipt of the start message, the node will start working as configured before (e.g. computing the function as defined during set up phase).
Into the start message payload, server side application can set two different flags, `radioAlwaysOn` and `enableTDMA`.

### `radioAlwaysOn`

When the `radioAlwaysOn` parameter is set to FALSE, the radio will be turned off if no message needs to be sent and turn on whenever the node has a message to send.
Since bidirectional communication must be still guarantee, the radio controller component implements the logic for listening to the messages the coordinator might be holding for it (Figure 3.3 and Figure 3.4).

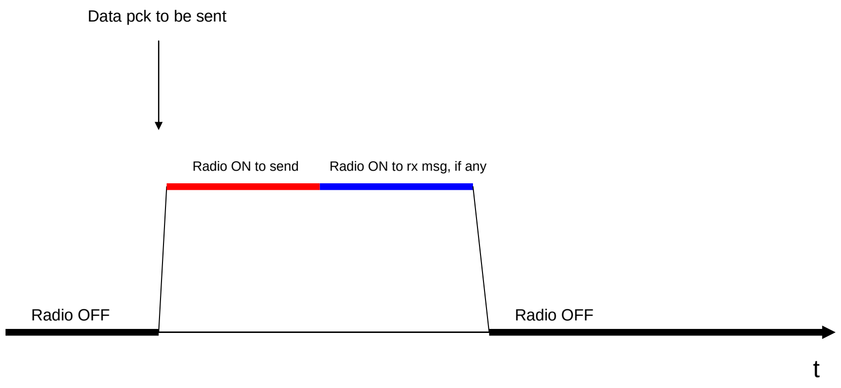

**Figure 3.3: radio behaviour when `radioAlwaysOn`=FALSE**

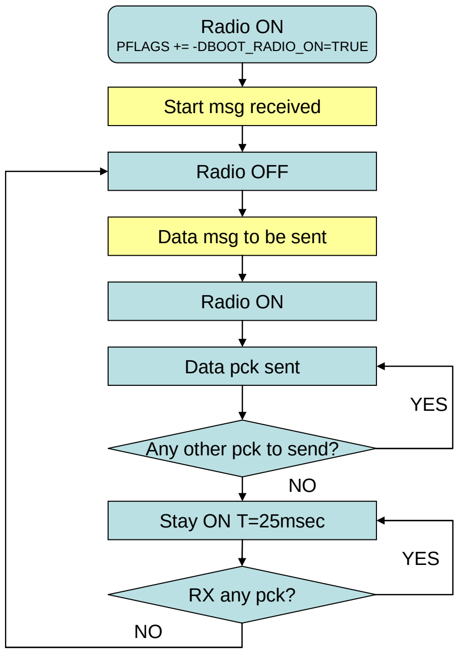

**Figure 3.4: flow chart of RadioController component when `radioAlwaysOn`=FALSE**

On the server side, if the network has been started up with `radioAlwaysOn = FALSE` and the SPINEManager has messages to send to a node, it stores the message until the next time it gets a data message form that node. Moreover, the coordinator waits for an acknowledgment from the node before deleting the message from the queue, otherwise it tries to retransmit it.The implementation of this mechanism might be found into the TOSLocalNodeAdapter.java.

## enableTDMA

When the `enableTDMA` parameter is set to TRUE, the Radio Controller will apply a TDMA schema to all its transmissions. Therefore, whenever it has to send a data packet it will wait for the next time slot that has been assigned to it. This may be needed in scenarios where a large amount of data must be sent to the coordinator from several nodes and a pure CSMA-CA schema could not be enough for achieving a reliable communication.
The start message will carry also the total number of nodes present into the network at that time. Upon receiption of the start message, if the TDMA is enabled, the node will allocate to its transmission a time slot that depends on its own ID and the total number of nodes into the network at that time. This is the reason why, users must flash nodes with sequential IDs (1,2,...) and, if needed, they can change the `TDMA_FRAME_PERIOD` into the Makefile, defined by default to be 600msec. Note that since the coordinator is not assigned to a particular time slot, it will try to send packets as soon as possible, relying on the default radio access schema of the base-station (typically CSMA-CA for TinyOS basestations).

The **Packet Manager** component takes care of building and parsing SPINE packet payload and header; moreover, if necessary, it takes care of the fragmentation of the packet before sending it to the radio. In this way, whenever the Function Manager has to send a packet, it does not have to care about the length and simply call `PacketManager.build`. Then the PacketManager will, if needed, parse the packet and send multiple packets. On the server side, the SPINE Manager will properly rebuild the packet. Note that, at this stage, the mechanism is not implemented in the other direction: therefore the server side will never send fragments of the same packet but will build messages according to the maximum length allowed.
The Packet Manager may take care of different types of incoming and outgoing packets as far as they implement the defined interfaces (`InPacket` with the parse command and `OutPacket` with the build command).

On the node side, communication interfaces might be found in the `tos\interfaces\communication` folder and their implementations in the `tos\system\communication` one.

On the server side, SPINE 1.2 core framework has been designed to be independent on the network it is communicating with (Figure 3.5). Therefore, all the SPINE Server Side core code contained into the `spine` and `spine.datamodel` packages do not use any TinyOS specific APIs.

TinyOS specific messages and low level communication APIs are used and defined into the `spine.communication.tinyos` package.

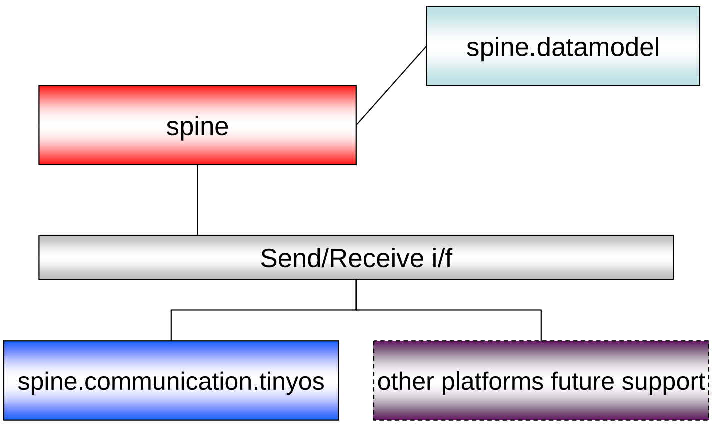

**Figure 3.5: SPINE 1.2 Server Side, architecture.**

SPINE1.2 defines a bidirectional protocol for communication between the coordinator node and the sensor nodes.

All the messages have the same SPINE protocol header that is build as illustrated below:

| Bit | 2 | 1 | 5 | 8 | 16 | 16 | 8 | 8 | 8 |
|---|---|---|---|---|---|---|---|---|---|
| | Version | Extension | Type | GroupID | SourceNodeID | DestNodeID | SequenceNr | Total fragments | Fragment seqNr |

In particular, it supports management messages from the coordinator to the nodes such as:

- Network Discovery. It is an empty packet, sent as broadcast so that can reach all the nodes into the network;
- Set Up Sensor. It contains the sampling time settings needed to set the sensor

| Bit | 4 | 2 | 2 | 16 |
|---|---|---|---|---|
| | Sensor Code | Time Scale | Reserved | Sampling Time |

- Set Up Function. It contains the general settings that are needed by a function. Parameters list is basically a list of byte so that can be very generic and used for different functions.

| Bit | 8 | 8 | Variable |
|---|---|---|---|
| | Function Code | Param Length | Param List |

- Activate/Deactivate Function. It contains details on the functionalities to be activated. Ad above, parameter list is a generic list of array.

| Bit | 8 | 8 | 8 | Variable |
|---|---|---|---|---|
| | Function Code | Activate/Deactivate | Param Len | Param List |

- Start Network. It is a broadcast packet that works as trigger to start the computation and the sampling on the nodes. It contains information about the total number of nodes present into the network (used if TDMA is enables) as well as flags for radio behaviors (duty cycling and TDMA).

| Bit | 16 | 8 | 8 |
|---|---|---|---|
|  | #nodes in the net | Radio always ON | Enable TDMA |

On the other hand, sensor nodes can send to the coordinator:

- Service Advertisement reports information about the sensors and the functions supported by the node.

| Bit | 8 | 8*#sensors | | 8 | Variable |
|---|---|---|---|---|---|
|  | #sensors | Sensor Code | Bitmask | #functions | Functions |

The function field is filled in by the specific Function and contains codes useful to indetify the function itself and other sub-functions if present.

- Data Packets with the function type and a list of bytes with the specific data. It has been defined to be very generic and may be easily used by all the other functionalities.

| Bit | 8 | 8 | Variable |
|---|---|---|---|
|  | Function Code | Param Len | Param List |

All SPINE TinyOS packets have been designed to be very general, therefore adding new sensors and new signal processing functionalities should not imply defining new messages.
If new messages are really needed, SPINE1.2 framework can be easily enhanced. Please contact the SPINE Team through the spine-dev@avalon.tilab.com mailing list for more details.

At this stage peer to peer communication among nodes is not supported yet, but its integration into the SPINE framework could be done very easily.

## 3.2 Sensing

The sensing part is composed by three main components: the Sensor Registry, the Sensor Board Controller and the Buffer Pool (Figure 3.6).

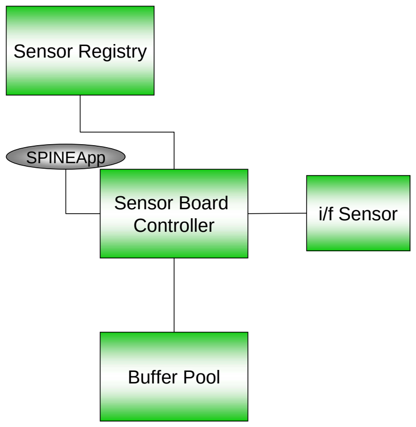

**Figure 3.6: SPINE 1.2 node side, Sensing Modules**

The **Sensor Board Controller** manages all the sensors that are registered, thanks to the **Sensor Registry**, to SPINE. Its main functionalities are setting sensor parameters (e.g. sampling time) and get data from them.
Data is then stored in a pool of buffers (**Buffer Pool**) that stores the data in a FIFO manner. Buffer Pool is set according to Makefile parameters `BUFFER_POOL_SIZE` and `BUFFER_LENGTH` to allocate a buffer for every channel of every registered sensor.
The sensing part may be enhanced by adding new sensors to be supported by SPINE.

### 3.2.1 How to introduce a new sensor

SPINE1.2 supports different kinds of sensors and different implementations of the same sensor (e.g accelerometer).
Adding a new sensor to the SPINE framework is quite easy. Sensor's driver must implement a defined Sensor interface to then be part of the SPINE framework and use all its functionalities.

### NODE SIDE

The steps to be followed to integrate your "`myNewSensor`" in SPINE1.2 TinyOS implementation on the node side are:

1. Look into `tos\interfaces\sensing` folder at `Sensor.nc`: this file contains the interface that your sensor driver must implement.

2. `tos\types\SensorConstants.h` must be updated to support a new sensor code `SensorCode` (`MY_NEW_SENSOR`)

3. Take one of the following approaches depending on how you answer this question: *do you already have a driver implemented to support your myNewSensor?*
   Both approaches might be used depending if you can change or write from scratch your driver implementation (approach a.) to make it according to SPINE 1.2 or you want to leave the driver implementation as it is and wrap it into the SPINE framework (approach b.).

a. *NO, I don’t have any driver already implemented* or *YES but I want to change it according to SPINE*. This means you now will write the driver for supporting your sensor according to Sensor SPINE interface. It is very simple: you’ll have to implement all the methods defined in `tos\interfaces\sensing\Sensor.nc`.
   SPINE1.2 supports, among others, a 3-axis accelerometer for the spine board. This is a good example to start with:
   i. Look into `tos\system\sensing` folder at `AccSensorC.nc`: this is the component that provides the platform independent layer of the AccSensor. You must create here a `MyNewSensorC.nc` file that links with the actual implementation of the driver (`HilMyNewSensorC`).
      If your sensor is already present among the ones supported by the current release of SPINE (e.g accelerometer), you do not have to implement this component, since it is already present (you can have a look to the actual implementation of the AccSensor both for the spine and the mts300 sensor boards).
   ii. `HilAccSensorC` can be found into `tos\sensorboards\spine` folder. Please note that **if you want to develop your own sensor board you can create a “mySensorBoard” folder that looks like spine folder with your drivers inside (`HilMyNewSensorC`)**
   iii. Having a look to `HilAccSensorP.nc`, you can see that it actually implements itself all the APIs defined in the `Sensor.nc` interface. This contains all the driver logic by itself. In your case will be `HilMyNewSensorP.nc` to implement the driver according to the defined APIs.

b. *YES and I need it as it is*: in this case you will not have to change your actual driver implementation but you will wrap it to match with the SPINE sensor interface. The SPINE gyroscope is a good reference implementation for this approach:
   i. Look into `tos\system\sensing` folder at `GyroSensorC.nc`: it is the same as above and you can see that the implementation is done by the `HilGyroSensorC` component (in your case `MyNewSensorC.nc` must be created).
      If your sensor is already present among the ones supported by the current release of SPINE (e.g accelerometer), you do not have to implement this component, since it is already present (you can have a look to the actual implementation of the AccSensor both for the spine and the mts300 sensor boards).
   ii. `HilGyroSensorC` can be found into `tos\sensorboards\spine` folder, that’s where `HilMyNewSensorC` should be put. Please note that **if you want to develop your own sensor board you can create a “mySensorBoard” folder that looks like spine folder with your drivers inside (`HilMyNewSensorC`)**
   iii. Look at `HilGyroSensorP.nc`, and notice that it implements all the APIs defined in the Sensor.nc interface wrapping already existing drivers (namely GyroXSensorC and GyroYSensorC) adding the logic to match with the defined interface.

4. In the implementation of the interface, in both cases, you need to take care of the registration to the SensorBoardRegistry. That’s very simple: in the Boot.booted() event implementation of the HilMySensorP.nc, you need to write:

```
call SensorsRegistry.registerSensor(MY_NEW_SENSOR);
```

5. If you defined a “mySensorBoard” you have to define it in the support/make/SPINE.extra

```
ifeq ($(SENSORBOARD),mySensorBoard)
     PFLAGS += -DMY_SENSOR_BOARD
endif
```

6. Now, you need to do some wiring. Go into `tos\system\sensing` folder and open `SensorBoardControllerC.nc`. You simply have to add 4 lines for each sensor supported by your sensor board. Those are needed basically to allow the SensorBoardController accessing the new sensor and reserving a timer for sampling it.

```
 #ifdef MY_SENSOR_BOARD
 /* For the myNewSensor Sensor */
      components myNewSensorC;
      components new TimerMilliC() as myNewSensorTimer;
      SensorBoardControllerP.SensorImpls[MY_NEW_SENSOR] -> myNewSensorC;
      SensorBoardControllerP.SamplingTimers[MY_NEW_SENSOR]                  ->
      myNewSensorTimer;
  #endif
```

   Indeed, if you add a new sensor to an existing sensor platform (e.g. spine), you'll have to add the wiring into the `#ifdef SPINE_SENSOR_BOARD`.

Please reference to comments into the Sensor.nc file to better understand what the interface methods have to provide.

## SERVER SIDE

Since SPINE1.2 server side provides APIs for managing on node functionalities, adding a new sensor support into the Java SPINE1.2 Server side, as far as it does not imply any new functionality, it is very simple.
You only have to add the `MY_NEW_SENSOR` constant into the `spine\SPINESensorConstants.java` as well as the string that defines it.
You'll only have to add 2 lines copying the ones for the accelerometer support here listed and setting the code and the string according your values.

```
public static final byte ACC_SENSOR = 0x01;

            case ACC_SENSOR: return ACC_SENSOR_LABEL;
```

Please note that `MY_NEW_SENSOR` value MUST be the same you set into the node side and MUST be different from the ones already supported by SPINE1.2, as listed in the constant file.

If your new sensor needs functionalities that are now not present in SPINE1.2, you'll have to add them as described in the following paragraph.

## 3.3 Processing

The processing part is composed by a function manager that takes care of all the functions supported on the node (Figure 3.7).

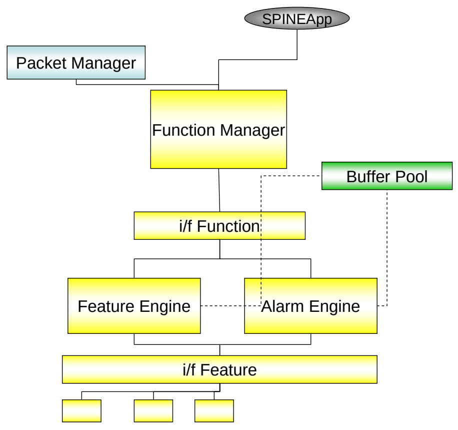

**Figure 3.7: SPINE 1.2 node side, Processing Modules**

On the node side, a generic function must implement the **Function** interface (tos\interfaces\processing) to be then managed by the **Function Manager**. The interface has been defined to be very general, in this way all the logic may be implemented inside the function.

The server Side sends two different commands to the Function Manager (setUpFunction and activateFunction) addressing a specific functions and sending a generic array of parameters that will then decoded by the implemented functionality. The data packet sent back to the Server Side with the computed values is composed as well by a generic array of bytes. The processing of the functions will then start upon receipt of the “start” message from the server Side.

Functions supported by SPINE1.2 release are Feature Engine and Alarm Engine.

### 3.3.1 Feature Engine

Data sampled by sensors on the motes may be sent to the coordinator without any processing (raw data) and then be analyzed on the server side. This may be not the most efficient procedure in terms of energy consumption and channel optimization.
Therefore, data may be processed on the node and only the result of the computation been sent to the coordinator.
SPINE 1.2 Feature Engine provides periodic calculation of simple feature on sensed data. The server side application can request a feature using two setup messages: a setup function message with values for the window and shift with a given sensor and an activate function message with the features desired for that sensor.
Features include AMPLITUDE, MAX, MEAN, MIN, MODE, PITCH&ROLL, RANGE, RAW DATA, RMS, STANDARD DEVIATION, TOTAL ENERGY, VARIANCE and VECTOR MAGNITUDE and can be calculated over multiple channels. For instance MEAN calculates the mean value over each active channel while VECTOR_MAGNITUDE is computed over all channels and returns a single value.

Feature engine can be configured by the serverSide application using setUpFunction and activateFunction messages.

**SETUP FEATURE ENGINE**

From the java application, the Feature Engine can be setup with different window and shift time for different sensors. For Feature Engine, we assume Window and Shift Size to be unique for a certain sensor (e.g. the same settings apply to all the channels of that sensor).

```
FeatureSpineSetupFunction ssf = new FeatureSpineSetupFunction();

ssf.setSensor(sensor);

ssf.setWindowSize(WINDOW_SIZE);

ssf.setShiftSize(SHIFT_SIZE);

manager.setupFunction(curr.getNodeID(), ssf);
```

**ACTIVATE FEATURES CALCULATION**

Once the feature engine is setup, several features can be activated to be computed on a single or multiple channels.
In the example below, the feature engine is set to compute the mode feature over the first channel and the minimum value over the second and the third channel of the sensor setup before.

```
FeatureSpineFunctionReq sfr = new FeatureSpineFunctionReq();

sfr.setSensor(sensor);

sfr.addFeature(SPINEFunctionConstants.MODE,CH1_ONLY);

sfr.addFeature(SPINEFunctionConstants.MIN,CH2_CH3_ONLY);

manager.activateFunction(curr.getNodeID(), sfr);
```

**FEATURE DATA**

Once the feature engine is set up and the needed features are activated, the Feature Engine waits for the "start" message to start computing the features. The Feature Engine will send back to the server application a feature packet every shift time. The Server Side will get, into the dataReceived event, a FeatureData packet that contains an array of Feature object. Therefore the application can access to the calculated Features values.

```
public void dataReceived(int  sourceNode, Data data) {
            switch (data.getFunctionCode()) {
                  case SPINEFunctionConstants.FEATURE: {
                        Feature[] feats = ((FeatureData)data).getFeatures();
                        Feature firsFeat = feats[0];
                        byte sensor = firsFeat.getSensorCode();
                        byte featCode = firsFeat.getFeatureCode();
                        int ch1Value = firsFeat.getCh1Value();
                        // do something with this feature data…
                        break;
                  }
                  default: break;
            }
}
```

### 3.3.1.1 How to add a new feature into the feature engine

SPINE1.2 feature engine supports all the features listed above but it can be easily enhanced to compute other features on the sensed data.

## NODE SIDE

1. In Spine_nodes1_2\tos\types\Functions.h add MY_NEW_FEATURE code into enum FeatureCodes
2. Into tos/system/processing you have to implement the feature logic. For this reason you'll create 2 new files: MyNewFeatureC.nc and MyNewFeatureP.nc.
   MyNewFeature module will have to provide the Feature interface (look into tos/interfaces/processing), meaning it will have to implement the .calculate and the .getResultSize commands.
   a. Feature.calculate command computes your feature based on the available data, which is a window length number of samples per each active channel.
   b. Feature.getResultSize returns the word size (e.g. 2 for uint16_t) for the array of elements which is written by the Feature.calculate command.

   Remind that, since the Boot interface is used, you'll have to implement the event Boot.booted(), where we suggest to register the feature to the FeatureEngine to be then discovered during discovery time.

   ```
   event void Boot.booted() {
      if (!registered) {
         // the feature self-registers to the FeatureEngine at boot time
         call FeatureEngine.registerFeature(SUM);
         registered = TRUE;
      }
   }
   ```

3. Open tos/system/processing/FeatureEngineC.nc and add the following lines:
   ```
   components MyNewFeatureC;
   FeatureEngineP.Features[MY_NEW_FEATURE] -> MyNewFeatureC;
   ```

## SERVER SIDE

Open `spine.SPINEFunctionConstants.java` and add the following lines that define the new feature:

```
public static final byte  MY_NEW_FEATURE =

public static final String MY_NEW_FEATURE_LABEL = "MyNewFeature"

case MY_NEW_FEATURE: return MY_NEW_FEATURE_LABEL;
```

The user must also add the new constants to switch statements of the methods in SPINEFunctionConstants.java (functionCodeByString, functionalityCodeToString).

## 3.3.2 Alarm Engine

Data sampled by sensors on the motes may be needed by the coordinator only if they respect certain conditions. Therefore conditions may be verified by the node itself and data sent to the coordinator only when needed: this is the aim of the SPINE Alarm Engine.
SPINE 1.2 Alarm Engine provides node-generated events whenever functions' values do not respect previously set thresholds. Alarms can be set on any of the supported features, including raw data, specifying window and shift settings as well as alarm-specific parameters.
The Alarm Engine provides notification for four different kinds of alarms on all the features available on the node (Figure 3.8).

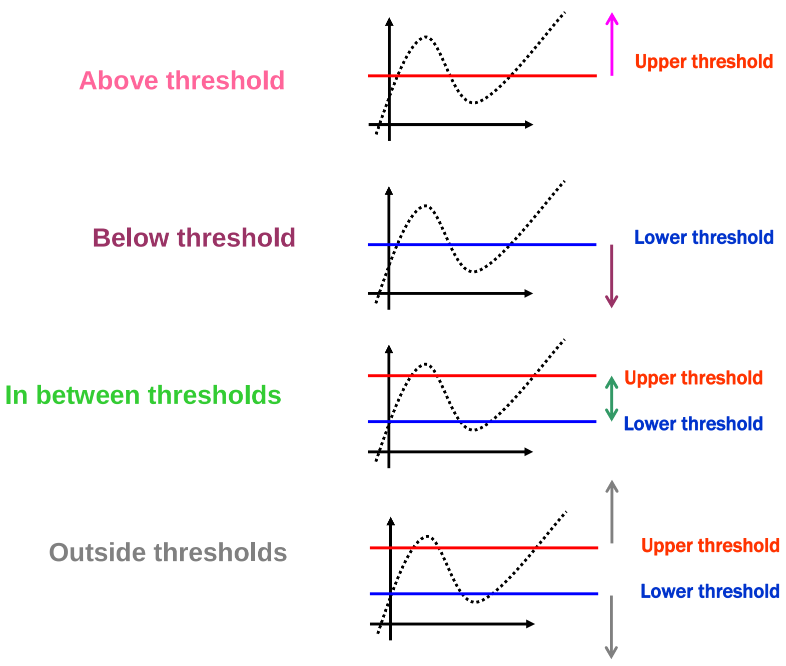

**Figure 3.8: Alarm types**

Alarm engine can be configured by the serverSide application using setUpFunction and activateFunction messages.

### SETUP ALARM ENGINE

From the java application, the Alarm Engine can be setup with different window and shift time for different sensors. As for the Feature Engine, we assume Window and Shift Size to be unique for a certain sensor (e.g. the same settings apply to all the channels of that sensor).

```
AlarmSpineSetupFunction ssf = new AlarmSpineSetupFunction();

ssf.setSensor(sensor);

ssf.setWindowSize(WINDOW_SIZE);

ssf.setShiftSize(SHIFT_SIZE);

manager.setupFunction(curr.getNodeID(), ssf);
```

## ACTIVATE ALARMS

Once the alarm engine is setup, different alarms can be activated. A single alarm is activated defining the kind of data it has to check (DataType), on which sensor (Sensor) and which channel (ValueType), thresholds (upper and lower) and which kind of alarm (AlarmType) to be pick among the four described in Figure 3.8.
In the example below, the Alarm Engine is configured to report the Maximum value of the accelerometer on Channel 1 (X axis) when it is above the upperThreshold.
Once the preset condition is verified, the node sends back to the coordinator all the information about the event that occurred.

```
AlarmSpineSetupFunction sfr = new AlarmSpineFunctionReq();

sfr.setDataType(SPINEFunctionConstants.MAX);

sfr.setSensor(SPINESensorConstants.ACC_SENSOR);

sfr.setValueType((SPINESensorConstants.CH1_ONLY));

sfr.setLowerThreshold(lowerThreshold);

sfr.setUpperThreshold(upperThreshold);

sfr.setAlarmType(SPINEFunctionConstants.ABOVE_Threshold);

manager.activateFunction(curr.getNodeID(), sfr);
```

## ALARM DATA

Once the Alarm Engine is set up and the requested alarms are activated, the Alarm Engine waits for the "start" message to start monitoring the data to check if an alarm condition is present.
The Alarm Engine will send back to the server application an alarm packet whenever the alarm condition occurs. The Server Side will get, into the dataReceived event, a AlarmData packet that contains information about the specific alarm.
Therefore the application can access to the values reported by the Alarm Engine.

```
public void dataReceived(int  sourceNode, Data data) {
            switch (data.getFunctionCode()) {
                  case SPINEFunctionConstants.ALARM: {
                        byte alarmType = ((AlarmData)data).getAlarmType();
                        int value = ((AlarmData)data).getCurrentValue();
                        byte sensor = ((AlarmData)data).getSensorCode();
                        // do something with this feature data…
                        break;
                  }
                  default: break;
            }
}
```

### 3.3.3 How to add a new function

## NODE SIDE

A generic function must implement the Function.nc (Spine_nodes1_2\tos\interfaces\processing\) interface and therefore provide the setUpFunction, activateFunction, disableFunction, getFuntionList, startComputing and stopComputing commands.
The inner implementation of these commands depends on the function logic and it's transparent to the SPINE core that communicates with all the functions, through the FunctionManager, with a general array of parameters.

Here few steps to integrate myNewFunction algorithm into SPINE1.2:

1. In Spine_nodes1_2\tos\types\Functions.h add MY_NEW_FUNCTION code into the enum FunctionCodesFunctionCodes
2. Open Spine_nodes1_2\tos\interfaces\processing\Function.nc: this is the interface that myNewFunction must implement.
   You have to place your implementation into the Spine_nodes1_2\tos\system\processing folder. In detail, your will write 2 files:
   a. MyNewFunctionEngineP.nc, module component that implements the function logic (e.g. classification algorithm...). This module has to implement the Function.nc interface as well as three events:
      i. `Boot.booted`
      ii. `FunctionManager.sensorWasSampledAndBuffered`
      iii. `BufferPool.newElem`
      Remind that into the `Boot.booted` event implementation you'll have to register the the new function to the function manager to be then announced during the discovery process. A typical implementation is:
```
event void Boot.booted() {
      if (!registered) {
            call FunctionManager.registerFunction(MY_NEW_FUNCTION);
            registered = TRUE;
      }
}
```
   b. MyNewFunctionEngineC.nc is the configuration component.

   Developing myNewFunction logic, whenever it is needed to send a data a packet to the coordinator, the FunctionManager.send command must be called with the MY_NEW_FUNCTION function code and the data payload array of bytes.
3. Then, open the Spine_nodes1_2\tos\system\processing\FunctionManagerC.nc and add the following lines:
   a. `components MyNewFunctionEngineC.;`
   b. `FunctionManagerP.Functions[MY_NEW_FUNCTION] -> MyNewFunctionEngineC;`

Now that your new function logic is implemented on the node side, you have to enhance the server side as well to be able to correctly activate the function and receive data from it.

## SERVER SIDE

SPINE1.2 provides two messages for setting on node functionalities from the server side. As all the other functions already present, myNewFunction developed on the node side, must be set using those messages:

1. Set up Function contains the function code and a generic list of bytes to set up the function.

| Bit | 8 | 8 | Param Length * 8 |
|---|---|---|---|
| | Function Code | Param Length | Param List |

You can define your own parameters according to the settings needed by your function.
On receipt of this message on the node side, parameters will be given to the function calling the setUpFunction method defined into the Function.nc interface.
You'll have to implement the encoding logic into `spine.communication.tinyos.MyNewFunctionSpineSetupFunction.java` that will extend the already defined SpineSetupFunction.java abstract class (refer to spine.communication.tinyos package for SpineSetupFunction.java)

2. Activate/Deactivate Function

   Depending on the functionality, you may need to activate (or deactivate) with specific settings subparts of this functionality. Hence, you'll have to fill in and send a activate/deactivate function packet.

| Bit | 8 | 8 | 8 | Variable |
|---|---|---|---|---|
| | Function Code | Activate/Deactivate | Param Len | Param List |

You can define your own parameters according to the settings needed by the activation of all or part of your function.
On receipt of this message on the node side, parameters will be given to the function calling the activateFunction method defined into the Function.nc interface.
You'll have to implement the encoding logic into `spine.communication.tinyos.MyNewFunctionSpineFunctionReq.java` that will extend the already defined `SpineFunctionReq.java` abstract class (refer to `spine.communication.tinyos` package for `MyNewFunctionSpineFunctionReq.java`).

Once myNewFunction has been set and activated, it will send data to the server side, therefore it should be able of decoding it.

3. Data Packets
   SPINE1.2 data packet format has been designed to be very general: it is basically an array of bytes which meaning change depending on the functionality.

| Bit | 8 | 8 | Variable |
|---|---|---|---|
| | Function Code | Param Len | Param List |

MyNewFunction implementation on the node side will send a data payload with proper fields that must be decoded at the server side.
When data is received as array of bytes at the server side, this array is passed to the decode method of SpineData class. This dynamically loads the implementation corresponding to the function type.
SPINE1.2 server side has been designed to be platform independent, which means that its core logic can run using different transport layers and protocols. For this reason, data formats must be defined in a platform independent way (into the spine.datamodel package) as well as in a platform specific way (in this release we support TinyOS2.x and the SPINE protocol on top of that).

Platform independent classes – into the spine.datamodel package:

a. MyNewFunction.java: this class represents the MyNewFunction entity and therefore contains a constructor, a toString and getters methods.
b. MyNewFunctionData.java: this class represents the MyNewFunctionData entity and t contains the decode method for converting low level MyNewFunction type data into an high level object.

Plaftorm dependent class – TinyOS2.x implementation into the spine.communication.tinyos package

c. MyNewFunctionSpineData.java: this class contains the static method to parse (decompress) a TinyOS SPINE MyNewFunction Data packet payload into a platform independent one. This class is invoked only by the SpineData class, thru the dynamic class loading. Note that this class is only used internally at the framework.