TinySchema: Creating Attributes and Commands in TinyOS

Wei Hong and Sam Madden
whong@intel-research.net, madden@cs.berkeley.edu

Version 0.1
August 21, 2002

1 Introduction

TinySchema is a collection of TinyOS components that manages a small respository of named attributes and commands that can be easily queried or invoked from inside or outside a mote network. A TinySchema *attribute* is much like a column in a traditional database system. It has a name and a type. In addition, TinySchema allows you to associate arbitrary TinyOS code to each attribute for getting and setting the attribute value. Once an attribute is created, it can be retrieved or updated through a unified interface provided by TinySchema. *TinyDB* (see TinyDB document), the in-network query processing system for TinyOS, is one of the applications built on top of this interface. You can also build your own application for manipulating attributes based on the interfaces provided by TinySchema. Typically, there are three classes of attributes:

- Sensor Attributes. These can be raw readings from sensors such as temperature and photo sensors, accelerometers, magnetometers, etc. They can also be computed sensor values after applying some calibration or signal processing logic.
- Introspective Attributes. These are values from internal software or hardware states, e.g., software version stamp, parent node in routing tree, battery voltage, etc. They are very useful for monitoring the health and statistics of a mote network.
- Constant Attributes. These are constant values assigned to a mote at programming time or run time, e.g., node id, group id, name, location, etc.

A TinySchema *command* is much like a stored procedure in a traditional database system. It consists of a name, a list of arguments and a return type. You can associate arbitrary TinyOS code to each command. TinySchema provides a unified interface for invoking these commands. TinyDB is also built on top of the TinySchema command interfaces for its trigger actions (see TinyDB document). Typically, there are two classes of commands:

- Actuation Command. These are commands that cause some physical actions on a mote, e.g., rebooting a mote, flash LEDs, sound buzzer, raise a blind (when connected to an appropriate actuator), etc.
- Tuning Command. These are commands that adjust internal parameters, e.g., routing policy, number of retransmissions, sample rate, etc.

Currently all attributes and all commands must be staticly built into each mote. We plan to integrate with the virtual machines being developed for the TinyOS such as Mate and Mottle to allow dynamic creation of attributes and commands.

TinySchema only runs on MICA motes with NesC.

2 System Overview

TinySchema has two separate components in `broken/dev/tos/lib`: *Attr.td* and *Command.td*. *Attr* provides all attributes related interfaces and *Command* provides all command related interfaces.

Attr provides the following interfaces:

- `StdControl` for initialization.
- `AttrRegister` for creating new attributes. It is parametrized by a `uint8_t` (for up to 256 such interfaces). Each non-constant attribute must be connected to one of these interfaces. The coding convention is not to hardwire a specific number when you wire to one these 256 interfaces, but to wire your interface to `Attr.Attr[unique("Attr")]` and let the NesC compiler to automatically choose a unique number for you.
- `AttrRegisterConst` for creating new constant attributes. It is a simplified interface of AttrRegister for attributes associated to constant values only.
- `AttrUse` for discovering and using attributes.

Command provides the following interfaces:

- `StdControl` for initialization.
- `CommandRegister` for creating new commands. It is parametrized by a `uint8_t` (for up to 256 such interfaces). Each command must be connected to one of these interfaces. The coding convention is not to hardwire a specific number when you wire to one these 256 interfaces, but to wire your interface to `Command.Cmd[unique("Command")]` and let the NesC compiler to automatically choose a unique number for you.
- `CommandUse` for discovering and using commands.

We will describe each of the above interfaces in details in the next section.

3 Detailed Interface Descriptions

3.1 Data Types and Error Codes

All of TinySchema's data types and error codes are defined in `broken/dev/tos/interfaces/SchemaTypes.h`. The following data types are supported:

- `VOID`: the void type. Used for defining commands that do not return anything.
- `INT8 and UINT8`: 8-bit signed and unsigned integer types.
- `INT16 and UINT16`: 16-bit signed and unsigned integer types.
- `INT32 and UINT32`: 32-bit signed and unsigned integer types.
- `TIMESTAMP`: not yet supported.
- `STRING`: null-terminated ASCII strings.
- `COMPLEX_TYPE`: not yet supported.

Here are the error codes used in all TinySchema interfaces:

- `SCHEMA_SUCCESS`: success!
- `SCHEMA_ERROR`: something is wrong.
- `SCHEMA_RESULT_READY`: the return result is ready in the result buffer. Used for non-split-phase attributes and commands.
- `SCHEMA_RESULT_NULL`: the return result is null.
- `SCHEMA_RESULT_PENDING`: the return result is not yet filled in in the result buffer. Must wait for the data ready event. Used for split-phase attributes and commands.

3.2 Attribute Related Interfaces

3.2.1 Attribute Data Structures

All attribute related data structures are defined in `broken/dev/tos/interfaces/Attr.h`. The main data structure is `AttrDesc` which contains the definition of each attribute. `AttrDescs` is just an array of `AttrDesc`'s for all the attributes defined in each mote. You must pay attention to the constants defined at the beginning of the file which defines the maximum number of attributes, maximum attribute name length, etc. Do not exceed those limits! Increase them as needed, but they cost more precious RAM space on a mote.

3.2.2 AttrRegister

```
command result_t registerAttr(char *name, TOSType attrType, uint8_t attrLen)
```

This is the command you call to register an attribute. The `attrLen` argument is only relevant to variable-length types such as STRING. It is ignored for fixed-length types.

```
event result_t getAttr(char *name, char *resultBuf, SchemaErrorNo *errorNo)
```

This is the TinyOS code that you must provide for getting the value of the attribute you just registered through `registerAttr`. `name` is the name of the attribute. It is mostly redundent, but may come in handy if you want to write one piece of code that supports mulitple attributes. `resultBuf` is a pointer to a pre-allocated buffer to hold the value of this attribute. You can assume that enough space has been allocated to hold the value of this attribute. `errorNo` is the return error code. You are required to do one of the following in `getAttr`:

- fill in the attribute value in `resultBuf` and set `*errorNo` to `SCHEMA_RESULT_READY`,
- or set `*errorNo` to `SCHEMA_RESULT_PENDING` and fill in `resultBuf` later when the data is ready and call `getAttrDone`,
- or set `*errorNo` to `SCHEMA_RESULT_NULL`,
- or set `*errorNo` to `SCHEMA_RESULT_ERROR`.

`event result_t setAttr(char *name, char *attrVal)`

This is the TinyOS code that you must provide for setting the value of the attribute you just registered through `registerAttr`. `name` is the name of the attribute. It is mostly redundent, but may come in handy if you want to write one piece of code that supports multiple attributes. `attrVal` is a pointer to a value of the same type as the attribute type. NULL pointer means a null value. If the value of this attribute cannot be set, simply return `FAIL`.

`command result_t getAttrDone(char *name, char *resultBuf, SchemaErrorNo errorNo)`

This is the command you must call for split-phase attributes. In this case, the `getAttr` will initiate a split-phase operation, set `*errorNo` to `SCHEMA_RESULT_PENDING` then return. In the split-phase completion event (e.g. `ADC.dataReady()`), you must call this command with the attribute value filled in `resultBuf`.

3.2.3 AttrRegisterConst

`command result_t registerAttr(char *name, TOSType attrType, char *attrVal)`

This command provides a simplified way to associate a constant value to an attribute without having to write the `getAttr` and `setAttr` code as described above in the `AttrRegister` interface. `attrVal` points to a value of the `attrType` type. The `Attr` component preallocates space to hold values for a fixed number (`MAX_CONST_ATTRS` defined in `broken/dev/tos/interfaces/Attr.h`) of constant attributes. This command assigns a slot in the preallocated space to hold the constant value at `attrVal`. The `AttrUse` interface to be described below will automatically handle the get and set of the newly defined constant attributes just like any other attributes. Currently, a constant attribute can be at most 4 bytes long.

3.2.4 AttrUse

`command AttrDescPtr getAttr(char *name)`

This command returns a pointer to the attribute descriptor for the attribute with a name that matches the argument. NULL will be returned if the attribute does not exist. The returned attribute descriptor is NOT to be freed.

`command AttrDescPtr getAttrById(uint8_t attrIdx)`

This command returns a pointer to the attribute descriptor corresponding to an attribute index.

`command uint8_t numAttrs()`

This command returns the total number of attributes that have been registered.

`command AttrDescsPtr getAttrs()`

This command returns the array of attribute descriptors for all the the attributes that have been registered.

`command result_t getAttrValue(char *name, char *resultBuf, SchemaErorNo *errorNo)`

This is the command retrieves the value of an attribute by name. `name` is the name of the attribute. `resultBuf` is a pointer to a preallocated buffer to hold the attribute value. It must be at least as big as the attribute length. `errorNo` is a return parameter of the error code. It has the following cases:

- SCHEMA_RESULT_READY. This means that the value of the attribute has already been copied into resultBuf. This is not a split-phase attribute.
- SCHEMA_RESULT_PENDING. This means that the attribute value is not ready. It will be ready when the getAttrDone event is signaled. This is a split-phase attribute.
- SCHEMA_RESULT_NULL. The value of this attribute is null.
- SCHEMA_RESULT_ERROR. Something is wrong.

```
command result_t setAttrValue(char *name, char *attrVal)
```

This command sets the value of an attribute by name. name is the attribute name. attrVal is a pointer to a value of the same type as the attribute. This command will return FAIL if the attribute cannot be set.

```
event result_t getAttrDone(char *name, char *resultBuf, SchemaErrorNo errorNo)
```

This event will be signaled after a getAttrValue command is called on a split-phase attribute when the value of the attribute is ready. By this time, the value of the attribute is already copied into resultBuf. errorNo are the same as described for getAttrValue.

3.3 Command Related Interfaces

3.3.1 Command Data Structures

All command related data structures are defined in broken/dev/tos/interfaces/Command.h. It defines the following important data structures:

- CommandDesc is for a command descriptor.
- CommandDescs is for an array of command descriptors.
- ParamList is a list of parameter types used in command definitions. There is a convinient varg function setParamList to populate a ParamList with a list of types.
- ParamVals is a a list of parameter values for command invocation.

You must pay attention to the constants defined at the beginning of Command.h for the current limitations such as maximum number of parameters in a command, maximum number of commands and maximum number of characters in a command name. These limits must be observed or extended at the cost of more RAM consumption.

3.3.2 CommandRegister

```
command result_t registerCommand(char *name, TOSType retType, uint8_t retLen, ParamList
*paramList)
```

This NesC command registers a new TinySchema command. name is the name of the command. retType is the return type of the command. Use the VOID type if the command does not return any

value. retLen is the maximum length for the return value for any variable length types such as STRING. It is ignored for fixed-length types. paramList is the list of parameter types that this command expects when invoked.

```
event result_t commandFunc(char *commandName, char *resultBuf, SchemaErrorNo *errorNo,
ParamVals *params)
```

This is the TinyOS code you provide that implements the command that you just registered through registerCommand. commandName is the name of the command. It is mostly redundant, but may come in handy when you want to write one piece of code to implement multiple commands. resultBuf is a pointer to the preallocated buffer this command's return value is supposed to be copied into. errorNo is the return parameter for error code. params is the list of parameter values for the current invocation. You are required to do one of the following in commandFunc:

- Non-split-phase return. Copy the return value to resultBuf, set *errorNo to SCHEMA_RESULT_READY or SCHEMA_RESULT_NULL then return.
- Split-phase return. Initiate the split-phase operation, set *errorNo to SCHEMA_RESULT_PENDING then return. commandDone must be called from the split-phase completion event.
- Error. Set *errorNo to SCHEMA_RESULT_ERROR then return.

```
command result_t commandDone(char *commandName, char *resultBuf, SchemaErrorNo errorNo)
```

This NesC command must be called in the split-phase completion event if commandFunc returns an error code of SCHEMA_RESULT_PENDING. commandName is the command name. resultBuf is a pointer to a buffer the return value is supposed to be copied into. errorNo is the error code.

3.3.3 CommandUse

```
command CommandDescPtr getCommand(char *name)
```

This NesC command looks up a TinySchema command descriptor by name. NULL is returned if the command does not exist.

```
command CommandDescPtr getCommandById(uint8_t idx)
```

This NesC command looks up a TinySchema command descriptor by index.

```
command uint8_t numCommands()
```

This NesC command returns the total number of TinySchema commands currently registered.

```
command CommandDescsPtr getCommands()
```

This NesC command returns an array of command descriptors for all the currently registered TinySchema commands.

```
command result_t invoke(char *commandName, char *resultBuf, SchemaErrorNo *errorNo, ParamVals
*params)
```

This NesC command is for invoking a TinySchema command. commandName is the name of the TinySchema command. resultBuf is a pointer to the buffer the return value is supposed to be copied into. errorNo is the return parameter for error code. ParamVals is the list of parameter values to be

passed into this TinySchema command. See the description of `getAttrValue` in Section 3.2.4 for all the error codes you should handle.

`command result_t invokeMsg(TOS_MsgPtr msg, char *resultBuf, SchemaErrorNo *errorNo)`

This NesC command is a wrapper over `invoke`. It first parses the `TOS_Msg` into a command name and a list of parameter values then calls `invoke`. `msg.data` is expected to start with the null-terminated string for command name followed by the list of parameter values tightly packed one after the other. This NesC command is introduced for supporting remote invocation of TinySchema commands via the radio.

`event result_t commandDone(char *commandName, char *resultBuf, SchemaErrorNo errorNo)`

This is the event you are supposed to implement to handle split-phase command completion. See the description for `getAttrDone` event in Section 3.2.4.

4 Examples

Directories `broken/dev/tos/lib/Attributes` and `broken/dev/tos/lib/Commands` contain all the ready-to-use components that implements the most common attributes and commands. These are also the attributes and commands that are built into TinyDB. They also serve as examples of TinySchema attribute and command implementations. The following is the list of files in these two directories and the corresponding attributes and commands that they implement. If you want to use any of these predefined attributes or commands in your application, simply wire the `StdControl` interface of these components to `Main.StdControl` and the attributes or commands will be automatically registered and ready to use.

- `broken/dev/tos/lib/Attributes/`
  - {`AttrAccel,AttrAccelM`}`.td` defines two attributes: *accel_x* and *accel_y* for the raw accelerometer readings in the X and Y axis respectively.
  - {`AttrGlobal,AttrGlobalM`}`.td` defines two attributes: *nodeid* and *group* for the node id and group id respectively.
  - {`AttrMag,AttrMagM`}`.td` defines two attributes: *mag_x* and *mag_y*. They are maximum magnetometer readings in the X or Y axis at 32 samples/second since the last time you get their values. At the same time, they also automatically adjust the X and Y potentiometers of the magnetometer to keep the readings centered and avoid railing. These two attributes are designed for detecting moving magnetic fields. For example, they are used in the car tracking demo in TinyDB in which the car (with a magnet) is detected by a mote based on spikes in the values of these two attributes.
  - {`AttrMic,AttrMicM`}`.td` defines four attributes: *rawmic*, *noise*, *rawtone* and *tones*. *rawmic* is the raw microphone ADC reading. *noise* is the maximum microphone reading at 32 samples/second since last time the attribute is read. *rawtone* returns 1 if a sounder tone is detected, 0 otherwise. *tones* returns the total number of tones detected at 32 samples/second since the last time this attribute is read.

- {AttrPhoto,AttrPhotoM}.td defines the *light* attribute. It returns the raw ADC reading from the photo sensor.
- {AttrPot,AttrPotM}.td defines the *pot* attribute. It returns the current potentiometer setting (transmit power). This attribute can also be set.
- {AttrTemp,AttrTempM}.td defines the *temp* attribute. It returns the raw temperature sensor reading.
- {AttrVoltage,AttrVoltageM}.td defines the *voltage* attribute. It returns the ADC reading for the current battery voltage. It is an indicator of how much battery power is remaining.

- broken/dev/tos/lib/Commands/
  - {CommandLeds,CommandLedsM}.td defines three commands: *SetLedR(UINT8)*, *SetLedG(UINT8)* and *SetLedY(UINT8)*. They control the Red, Green and Yellow LEDs on a mote respectively. An argument of 0 means turning the LED off, 1 means on and 2 means toggle. All three commands return VOID.
  - {CommandPot,CommandPotM}.td defines the *SetPot(UINT8)* command. It sets the potentiometer value (transmit power) on a mote.
  - {CommandReset,CommandResetM}.td defines the dangerous *reset* command. It reboots a mote.
  - {CommandSounder,CommandSounderM}.td defines the *SetSnd(INT16)* command. It turns on the buzzer for a period specified by the argument (in milliseconds).