Storage

From TinyOS Wiki

This lesson introduces permanent (non-volatile) data storage in TinyOS. Permanent storage allows a node to persist data even if power is disconnected or the node is reprogrammed with a new image. You will become familar with different kinds of data storage including small objects, logs, and large objects. You will be exposed to the TinyOS interfaces and components that support permanent data storage on motes and you will learn how to:

- Divide the flash chip into volumes, which allows multiple and/or different type of data to be stored.
- Store configuration data that survives a power cycle.
- Store packets using the logging abstraction and retransmit the overheard packets after a power cycle.

Contents

Introduction

TinyOS 2.x provides three basic storage abstractions: small objects, circular logs, and large objects. TinyOS 2.x also provides *interfaces* to abstract the underlying storage services and *components* that *provide* (implement) these interfaces.

Interfaces

Let's take a look at some of the interfaces that are in the `tos/interfaces` directory and the types defined in the `tos/types` to familiarize ourselves with the general functionality of the storage system:

- `BlockRead (http://www.tinyos.net/tinyos-2.x/tos/interfaces/BlockRead.nc)`
- `BlockWrite (http://www.tinyos.net/tinyos-2.x/tos/interfaces/BlockWrite.nc)`
- `Mount (http://www.tinyos.net/tinyos-2.x/tos/interfaces/Mount.nc)`
- `ConfigStorage (http://www.tinyos.net/tinyos-2.x/tos/interfaces/ConfigStorage.nc)`
- `LogRead (http://www.tinyos.net/tinyos-2.x/tos/interfaces/LogRead.nc)`
- `LogWrite (http://www.tinyos.net/tinyos-2.x/tos/interfaces/LogWrite.nc)`
- `Storage.h (http://www.tinyos.net/tinyos-2.x/tos/types/Storage.h)`

Components

Components provide concrete implementations of the interfaces. You should be familiar with these components because your code needs to specify both the *interfaces* your application *uses* as well as the *components* which *provide* (implement) these interfaces:

- `ConfigStorageC (http://www.tinyos.net/tinyos-2.x/tos/chips/stm25p/ConfigStorageC.nc)`
- `LogStorageC (http://www.tinyos.net/tinyos-2.x/tos/chips/stm25p/LogStorageC.nc)`
- `BlockStorageC (http://www.tinyos.net/tinyos-2.x/tos/chips/stm25p/BlockStorageC.nc)`

Implementations

The preceding components are actually *chip-specific implementations* of these abstractions. Since TinyOS supports multiple platforms, each of which might have its own implementation of the storage drivers, you may need to be aware of platform-specific constants or other details (e.g. erase size) that can couple a storage client to the underlying chip-specific implementation.

For example, the preceding links are all specific to the ST Microelectronics M25Pxx family of flash memories used in the Telos and Tmote Sky motes. You *do not* need to worry about the details of where these files reside because TinyOS's make system includes the correct drivers automatically. However, you *do* need to know what these components are called because your code must list them in a `components` declaration.

If you are curious, the following links will let you browse the implementations of the Atmel AT45DB family of flash memories used in the Mica2/MicaZ motes:

- `ConfigStorageC (http://www.tinyos.net/tinyos-2.x/tos/chips/at45db/ConfigStorageC.nc)`
- `LogStorageC (http://www.tinyos.net/tinyos-2.x/tos/chips/at45db/LogStorageC.nc)`
- `BlockStorageC (http://www.tinyos.net/tinyos-2.x/tos/chips/at45db/BlockStorageC.nc)`

Finally, the following links will let you browse the implementation for the Intel imote2 flash memory:

- `ConfigStorageC (http://www.tinyos.net/tinyos-2.x/tos/platforms/intelmote2/ConfigStorageC.nc)`
- `LogStorageC (http://www.tinyos.net/tinyos-2.x/tos/platforms/intelmote2/LogStorageC.nc)`
- `BlockStorageC (http://www.tinyos.net/tinyos-2.x/tos/platforms/intelmote2/BlockStorageC.nc)`

Volumes

TinyOS 2.x divides a flash chip into one or more fixed-sized *volume*s that are specified at compile-time using an XML file. This file, called the volume table, allows the application developer to specify the name, size, and (optionally) the base address of each volume in the flash. Each volume provides a single type of storage abstraction (e.g. configuration, log, or block storage). The abstraction type defines the physical layout of data on the flash memory. A volume table might look like:

```
<volume_table>
  <volume name="CONFIGLOG" size="65536"/>
  <volume name="PACKETLOG" size="65536"/>
  <volume name="SENSORLOG" size="131072"/>
  <volume name="CAMERALOG" size="524288"/>
</volume_table>
```

The volume table for a particular application must be placed in the application's directory (where one types 'make') and must be named `volumes-CHIPNAME.xml` where CHIPNAME is replaced with the platform-specific flash chip's name. For example, the Telos mote uses the ST Microelectronics M25P family of flash memories. The drivers for these chips can be found in the `tos/chips/stm25p` directory. Therefore, a Telos-based application that uses the storage abstractions needs a file named `volumes-stm25p.xml`.

Note that the size parameter is a multiple of the erase unit for a particular flash chip. See Section 4.1 in TEP 103 <ref name="fn1">TEP 103: Permanent Data Storage (http://www.tinyos.net/tinyos-2.x/doc/html/tep103.html) </ref> for more details.

Storing Configuration Data

This lesson shows how configuration data can be written to and read from non-volatile storage. Configuration data typically exhibit some subset of the following properties. They are **limited in size**, ranging from a fews tens to a couple hundred bytes. Their values may be **non-uniform** across nodes. Sometimes, their values are **unknown** prior to deployment in the field and sometimes their values are **hardware-specific**, rather than being tied to the software running on a node.

Because configuration data can be non-uniform across nodes or unknown *a priori*, their values may be difficult to specify at compile-time and since the data are sometimes hardware-specific, their values must survive reprogramming, suggesting that encoding these values in the program image is not the simplest approach. Storing configuration data in volatile memory is also problematic since this data would not survive a reset or power cycle.

In summary, configuration data must persist through node resets, power cycles, or reprogramming, and then be restored afterward. The ability to persist and restore configuration data in this manner is useful in many scenarios.

- **Calibration.** Calibration coefficients for sensors might be factory-configured and persisted, so they are not lost when power is removed for shipping or the node is reprogrammed post-calibration. For example, a hypothetical temperature sensor might have an offset and gain that must be calibrated, because these parameters are hardware-specific, and stored because they are needed to convert the output voltage into the more useful units of degrees Celcius. The calibration data for such a sensor might look like:

```
typedef struct calibration_config_t {
  int16_t temp_offset;
  int16_t temp_gain;
} calibration_config_t;
```

- **Identification.** Device identification information, like IEEE-compliant MAC addresses or the TinyOS TOS_NODE_ID parameters are non-uniform across nodes although they are not hardware-specific, once they are assigned to a node, these values should be *sticky* in that they are persisted across reset, power cycle, and reprogramming operations (and not lost or reassigned to another node).

```
typedef struct radio_config_t {
  ieee_mac_addr_t mac;
  uint16_t tos_node_id;
} radio_config_t;
```

- **Location.** Node location data may be unknown at compile-time and only become available during deployment. An application might, for example, store node coordinates as follows and update these values in the field:

```
typedef struct coord_config_t {
  uint16_t x;
  uint16_t y;
  uint16_t z;
} coord_config_t;
```

- **Sensing.** Sensing and signal processing parameters like sample period, filter coefficients, and detection thresholds might be adjusted in the field. The configuration data for such an application might look like:

```
typedef struct sense_config_t {
  uint16_t temp_sample_period_milli;
```

```
  uint16_t temp_ema_alpha_numerator;
  uint16_t temp_ema_alpha_denominator;
  uint16_t temp_high_threshold;
  uint16_t temp_low_threshold;
} sense_config_t;
```

Now that we have discussed *why* one might use this type of storage, let's see *how* to use it. We will implement a simple demo application that illustrates how to use the `Mount` and `ConfigStorage` abstractions. A timer period will be read from flash, divided by two, and written back to flash. An LED is toggled each time the timer fires. But, before diving into code, let's discuss some high-level design considerations.

See `tinyos-2.x/apps/tutorials/BlinkConfig/` (http://www.tinyos.net/tinyos-2.x/apps/tutorials/BlinkConfig/) for the accompanying code.

Prior to its first usage, a volume does not contain any valid data. So, our code should detect the first usage of a volume and take any appropriate actions (e.g. preload it with default values). Similarly, when the data layout of the volume changes (for example, if the application requires new or different configuration variables), then application code should detect this and take appropriate actions (e.g. migrate the old data to the new layout or erase the volume and reload the defaults). These requirements suggest that we should have a way of keeping track of the volume version. We will use a version number for this purpose (and will need to maintain a discipline of updating the version number when the data layout changes incompatibly). Our configuration struct might have the following fields for the version number and blink period:

```
typedef struct config_t {
  uint16_t version;
  uint16_t period;
} config_t;
```

1. Create a `volumes-CHIPNAME.xml` file, enter the volume table in this file, and place the file in the application directory. Note that `CHIPNAME` is the flash chip used on your target plaform. For example, `CHIPNAME` will be `stm25p` for the Telos platform and `at45db` for the MicaZ platform. Our file will have the following contents:

   ```
   <volume_table>
     <volume name="LOGTEST" size="262144"/>
     <volume name="CONFIGTEST" size="131072"/>
   </volume_table>
   ```

   This volume information is used by the toolchain to create an include file. The auto-generated file, however, has to be included manually. Place the following line in the configuration file which declares the ConfigStorageC component (e.g. `BlinkConfigAppC.nc`):

   ```
   #include "StorageVolumes.h"
   ```

2. BlinkConfigC, the application code for this simple demo, *uses* the `Mount` and `ConfigStorage` interfaces (note that we rename `ConfigStorage` to `Config`).

   ```
   module BlinkConfigC {
     uses {
       ...
       interface ConfigStorage as Config;
       interface Mount;
       ...
     }
   }
   ```

3. Each interface must be wired to an *implementation* that will provide it:

```
configuration BlinkConfigAppC {
}
implementation {
  components BlinkConfigC as App;
  components new ConfigStorageC(VOLUME_CONFIGTEST);
  ...

  App.Config     -> ConfigStorageC.ConfigStorage;
  App.Mount      -> ConfigStorageC.Mount;
  ...
}
```

4. Before the flash chip can be used, it must be mounted using the two-phase mount/mountDone command. Here we show how this might be chained into the boot sequence:

```
  event void Boot.booted() {
    conf.period = DEFAULT_PERIOD;

    if (call Mount.mount() != SUCCESS) {
      // Handle failure
    }
  }
```

5. If the Mount.mount succeeds, then the `Mount.mountDone` event will be signaled. The following code shows how to check if the volume is valid, and if it is, how to initiate a read from the volume using the `ConfigStore.read` command. If the volume is invalid, calling `Config.commit` will make it valid (this call is also used to flush buffered data to flash much like the UNIX fsync system call is supposed to flush buffered writes to disk):

```
  event void Mount.mountDone(error_t error) {
    if (error == SUCCESS) {
      if (call Config.valid() == TRUE) {
        if (call Config.read(CONFIG_ADDR, &conf, sizeof(conf)) != SUCCESS) {
          // Handle failure
        }
      }
      else {
        // Invalid volume.  Commit to make valid.
        call Leds.led1On();
        if (call Config.commit() == SUCCESS) {
          call Leds.led0On();
        }
        else {
          // Handle failure
        }
      }
    }
    else{
      // Handle failure
    }
  }
```

6. If the read is successful, then a `Config.readDone` event will occur. In this case, we first check for a successful read, and if successful, we then check the version number. If the version number matches what we expected, we copy of the configuration data to a local variable, and adjust its values. If there is a version mismatch, we set the value of the configuration information to a default value. Finally, we call the the `Config.write` function:

```
  event void Config.readDone(storage_addr_t addr, void* buf,
    storage_len_t len, error_t err) __attribute__((noinline)) {

    if (err == SUCCESS) {
      memcpy(&conf, buf, len);
      if (conf.version == CONFIG_VERSION) {
        conf.period = conf.period/2;
        conf.period = conf.period > MAX_PERIOD ? MAX_PERIOD : conf.period;
        conf.period = conf.period < MIN_PERIOD ? MAX_PERIOD : conf.period;
      }
      else {
        // Version mismatch. Restore default.
```

```
      call Leds.led1On();
      conf.version = CONFIG_VERSION;
      conf.period = DEFAULT_PERIOD;
    }
    call Leds.led0On();
    call Config.write(CONFIG_ADDR, &conf, sizeof(conf));
  }
  else {
    // Handle failure.
  }
}
```

7. Data is not necessarily "written" to flash when `ConfigStore.write` is called and `Config.writeDone` is signaled. To ensure data is persisted to flash, a `ConfigStore.commit` call is required:

```
event void Config.writeDone(storage_addr_t addr, void *buf,
  storage_len_t len, error_t err) {
  // Verify addr and len

  if (err == SUCCESS) {
    if (call Config.commit() != SUCCESS) {
      // Handle failure
    }
  }
  else {
    // Handle failure
  }
}
```

8. Finally, when the `Config.commitDone` event is signaled, data has been durably written to flash and will survive a node power cycle:

```
event void Config.commitDone(error_t err) {
  call Leds.led0Off();
  call Timer0.startPeriodic(conf.period);
  if (err != SUCCESS) {
    // Handle failure
  }
}
```

Logging Data

Reliable (atomic) logging of events and small data items is a common application requirement. Logged data should not be lost if a system crashes. Logs can be either linear (stop logging when the volume is full) or circular (overwrite the least recently written data when the volume is full).

The TinyOS LogStorage abstraction supports these requirements. The log is record based: each call to LogWrite.append (see below) creates a new record. On failure (a crash or power cycle), the log only loses whole records from the end of the log. Additionally, once a circular log wraps around, log writes only lose whole records from the beginning of the log.

A demo application called `PacketParrot` shows how to use the `LogWrite` and `LogRead` abstractions. A node writes received packets to a circular log and retransmits the logged packets (or at least the parts of the packets above the AM layer) when power is cycled.

See `tinyos-2.x/apps/tutorials/PacketParrot/` (http://www.tinyos.net/tinyos-2.x/apps/tutorials/PacketParrot/) for the accompanying code.

The application logs packets it receives from the radio to flash. On a subsequent power cycle, the application transmits any logged packets, erases the log, and then continues to log packets again. The red LED is on when the log is being erased. The blue (yellow) LED turns on when a packet is received and turns off when a packet

has been logged successfully. The blue (yellow) LED remains on when packets are being received but are not logged (because the log is being erased). The green LED flickers rapidly after a power cycle when logged packets are transmitted.

1. The first step when using the log is to decide what kind of data you want to store in the log. In this case, we will declare a struct of the type:

```
typedef nx_struct logentry_t {
  nx_uint8_t len;
  message_t msg;
} logentry_t;
```

2. Unlike Config storage, Log storage does not require the volume to be explicitly mounted by the application. Instead, a simple read suffices in which a buffer and the number of bytes to read are passed to `LogRead.read`:

```
event void AMControl.startDone(error_t err) {
  if (err == SUCCESS) {
    if (call LogRead.read(&m_entry, sizeof(logentry_t)) != SUCCESS) {
      // Handle error
    }
  }
  else {
    call AMControl.start();
  }
}
```

3. If the call to `LogRead.read` returns SUCCESS, then a `LogRead.readDone` event will be signaled shortly thereafter. When that happens, we check if the data that was returned is the same length as what we expected. If it is, we use the data but if not, we assume that either the log is empty or that we have lost synchronization, so the log is erased:

```
event void LogRead.readDone(void* buf, storage_len_t len, error_t err) {
  if ( (len == sizeof(logentry_t)) && (buf == &m_entry) ) {
    call AMSend.send[call AMPacket.type(&m_entry.msg)](call AMPacket.destination(&m_entry.msg), &m_entry
    call Leds.led1On();
  }
  else {
    if (call LogWrite.erase() != SUCCESS) {
      // Handle error.
    }
    call Leds.led0On();
  }
}
```

4. The `PacketParrot` application stores packets received over the radio to flash by first saving the `message_t` and its length to a `log_entry_t` struct and then calling `LogWrite.append`:

```
event message_t* Receive.receive[uint8_t id](message_t* msg, void* payload, uint8_t len) {
 call Leds.led2On();
  if (!m_busy) {
    m_busy = TRUE;
    m_entry.len = len;
    m_entry.msg = *msg;
    if (call LogWrite.append(&m_entry, sizeof(logentry_t)) != SUCCESS) {
      m_busy = FALSE;
    }
  }
  return msg;
}
```

5. If the `LogWrite.append` returned SUCCESS, then a short time later, a `LogWrite.appendDone` will be signaled. This event returns the details of the write including the source buffer, length of data written, whether any records were lost (if this is a circular buffer) and any error code. If no errors occurred, then

the data was written to flash with atomicity, consistency, and durability guarantees (and will survive node crashes and reboots):

```
event void LogWrite.appendDone(void* buf, storage_len_t len,
                               bool recordsLost, error_t err) {
  m_busy = FALSE;
  call Leds.led2Off();
}
```

Storing Large Objects

Block storage is generally used for storing large objects that cannot easily fit in RAM. Block is a low-level system interface that requires care when using since it is essentially a write-once model of storage. Rewriting requires an erase which is time-consuming, occurs at large granularity (e.g. 256 B to 64 KB), and can only happen a limited number of times (e.g. 10,000 to 100,000 times is typical). The TinyOS network reprogramming system uses Block storage to store program images.

See `tinyos-2.x/apps/tests/storage/Block/` (http://www.tinyos.net/tinyos-2.x/apps/tests/storage/Block/) for an example of code that uses the Block storage abstraction.

Conclusions

This lesson introduced the basic storage abstractions in Tiny 2.x.

Related Documentation

<references/> Getting Started with TinyOS and nesC *TinyOS Programming*

< **Previous Lesson** | **Top** | **Next Lesson** >