Incremental Network Programming for Wireless Sensors

Jaein Jeong

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2005-17
http://www.eecs.berkeley.edu/Pubs/TechRpts/2005/EECS-2005-17.html

November 21, 2005

Incremental Network Programming for Wireless Sensors

by Jaein Jeong

Research Project

Submitted to the Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, in partial satisfaction of the requirements for the degree of **Master of Science, Plan II**.

Approval for the Report and Comprehensive Examination:

Committee:

Professor Kristofer S.J. Pister
Research Advisor

(Date)

* * * * * * *

Professor David E. Culler
Second Reader

(Date)

Incremental Network Programming for Wireless Sensors

Abstract

Incremental Network Programming for Wireless Sensors

by

Jaein Jeong

Master of Science in Computer Science

University of California at Berkeley

Professor Kristofer S.J. Pister, Research Advisor

One problem of the current network programming implementation of TinyOS is that it takes much longer than the traditional in-system programming due to the slow radio connection. We extended the network programming implementation of TinyOS 1.1 release so that it can reduce the programming time using the program code history. The host program generates the difference of the two program images using the Rsync algorithm and it sends the difference to the sensor nodes as script messages. The sensor nodes rebuild the program image based on the previous program version and the received script messages. The Rsync algorithm compares the two binary files and finds the matching blocks even though they are located in an arbitrary location within the files. We were able to get speed up of 9.1 for changing a constant and 2.1 to 2.5 for chaning a few lines in the source code.

Contents

List of Figures

List of Tables

Acknowledgements

Special thanks to Professor David Culler who gave valuable comments on this work. Thanks to Crossbow Technology for providing the source code for the network programming module and the boot loader. This work was supported in part by DARPA NEST Contract F33615-01-C1895.

Chapter 1

Introduction

Typically, wireless sensors are designed for low power consumption and small size and they don't have enough computing power and storage to support the rich programming development environment. Thus, the program code is developed in a more powerful host machine and is loaded to a sensor node afterwards. The program code is usually loaded to a sensor node through the parallel or serial port that is directly connected to the host machine and this is called in-system programming (Figure 1.1). In-system programming (ISP) is the most common way of programming sensor nodes because most microcontrollers support program loading with the parallel or serial port. However, ISP can load the program code to only one sensor node at a time. The programming time increases proportional to the number of wireless sensors to be deployed.

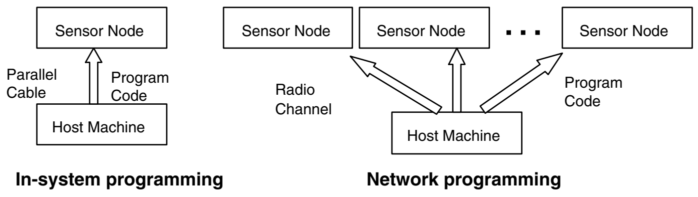


Figure 1.1: Comparison of network programming with in-system programming.

During the development cycle of wireless sensor software, the source code can be modified for bug fixes for additional functionalities. With ISP, the cost of software update is high; it involves all the efforts of collecting the sensor nodes placed in different locations and possibly disassembling and reassembling the enclosures. Network programming reduces these efforts by delivering the program code to each of the sensor nodes through the wireless links (Figure 1.1).

The network programming has been used with the introduction of TinyOS 1.1 release [1], [2]. This implementation, XNP (Crossbow Network Programming), provides the basic capability of the network programming; it delivers the program code to the sensor nodes remotely. However, it has some limitations:

First, XNP does not scale to a large sensor network. XNP disseminates program code only to the nodes that can be directly reached by the host machine. Thus, the nodes outside the single hop boundary cannot be programmed.

Second, XNP has low effective bandwidth compared to ISP. An experiment in [1] shows the programming time of XNP and ISP. In the experiment, we used a simple test application 'XnpCount' which has basic functionalities: network programming, counting numbers in LEDs and transmitting the numbers in radio packets. The version of 'XnpCount' we used is 37,000 bytes in size and it requires 841 XNP packets to transfer the whole program. The programming time of XNP was more than 4 times longer than that of ISP (Figure 1.2).

When XNP updates the program code with another version, it sends the whole program code rather than the difference. This incurs the same programming time even when the difference is small. If the sensor nodes can build program code image incrementally using the previous code image, the overall programming time can be reduced.

Our idea is to achieve fast code delivery by transmitting the difference between the two versions.

The rest of the thesis is organized as follows. Chapter 2 describes the in-system

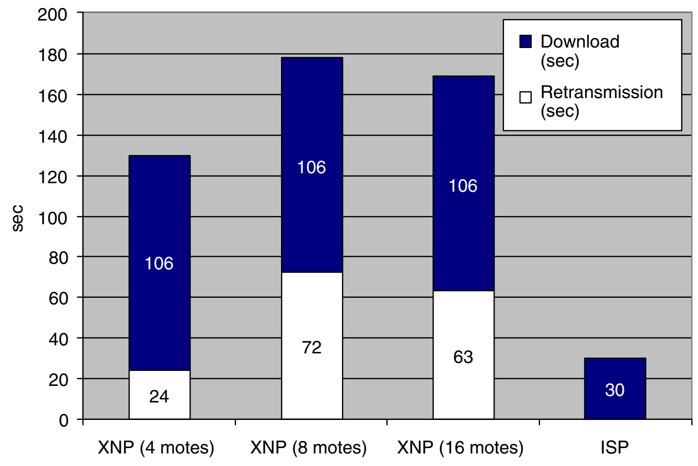


Figure 1.2: Programming time of XNP and ISP

programming and the network programming as a background. Chapter 3 outlines the incremental network programming and explains our first implementation. In Chapter 4, we use the Rsync algorithm to generate the program and show how this implementation improves the performance. In Chapter 5, we discuss the extension to the script delivery which makes the program delivery more reliable and faster. Chapter 6 discusses the related work on wireless sensor network programming. Finally, we conclude this thesis with Chapter 7.

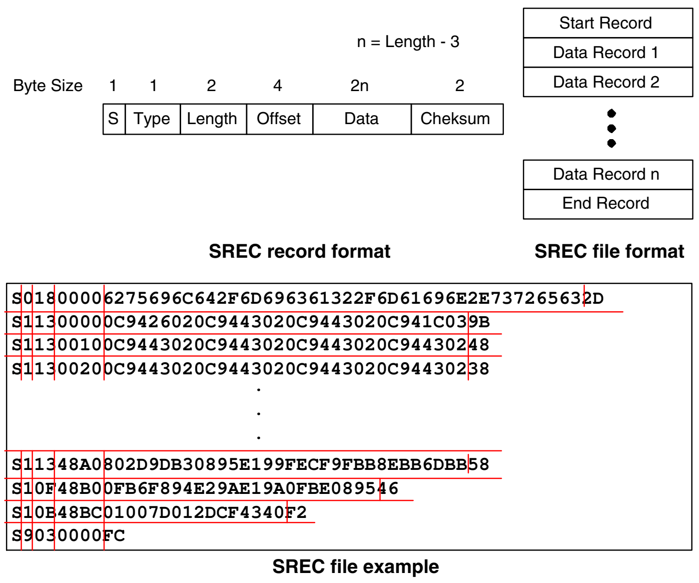
Chapter 2

Background

2.1 In-System Programming

The program development for wireless sensors starts with writing the source code. In UC Berkeley sensor platform, the source code is written in nesC programming language. Once the source code is successfully compiled, the binary code is generated (`main.exe`). The binary code is further converted to Motorola SREC format (`main.srec`) and is available for loading. Motorola SREC format is an ASCII representation of binary code and each line of an SREC file contains the data bytes of the binary code with additional house keeping information (Figure 2.1).

With ISP, the binary code (SREC format) is loaded to a sensor node through the direct connection (e.g. parallel port) from the host machine. The host programming tool (uisp) sends a special sequence of bytes that leaves the microcontroller of the sensor node in the programming mode. While the microcontroller is in programming mode, the data bytes sent by the host programming tool are directly written to the program memory of the microcontroller (Figure 2.2).



```
S0180000627569 6C642F6D696361322F6D61696E2E737265632D
```

Figure 2.1: Format of SREC file and its records with an example

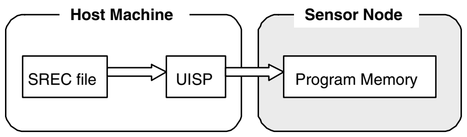


Figure 2.2: Process of in-system programming

2.2 Network Programming

Network programming takes a different approach to load the program code. Rather than write program code directly to the program memory, network programming loads program code in two steps. First, it delivers the program code to sensor nodes. Second, it makes the sensor nodes move the downloaded code to the program memory (Figure 2.3).

In the first step, the network programming module stores the program code in the external storage. Since the network programming module runs in the user level as a part of the main application code, it does not have the privilege to write program code into the program memory. In case of XNP, the network programming module writes the program

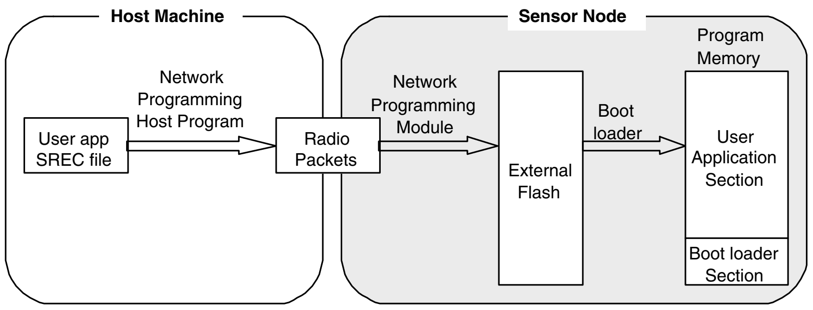


Figure 2.3: Process of network programming

code in the external flash memory outside the program memory. The external flash memory of MICA2/MICA2DOT motes is 512KB in size and it is big enough for any application code (the maximum size of 128KB). During the program delivery, part of the code can be missing due to the packet loss. The network programming module requests any missing records of the program code to make sure that there are no missing records.

In the second step, the boot loader copies the program code in the external flash memory to the program memory. The boot loader is a program that resides in the high memory area (which we call the boot loader section) of ATmega128 microcontroller and has the privilege to write data bytes to the user application section of the program memory [16]. The boot loader starts execution when it is called by the network programming module. After it copies the program code from the external flash memory to the program memory, it restarts the system.

In the paragraphs above, we assumed that the sensor nodes can update the current program image through the network programming. However, a sensor node cannot be network programmed until it has the network programming module and the boot loader. Thus, we need to load the initial program code and the boot loader with ISP.

For network programming, the host machine and the sensor nodes exchange XNP packets. An XNP packet has Active Message ID of 47 and its function (e.g. start download, download, query and reprogram) is specified in the command field of the packet.

Chapter 3

Design and Implementation

To design an incremental network programming mechanism, we need to consider some factors that affect the performance. Compared to other sensor applications, network programming keeps a large amount of data in sensor nodes and this contributes to the long programming time. Since the programming time is proportional to the data size, reducing the amount of transmission data will improve the programming time. External flash memory which is used for the program storage also limits the performance. The downloaded code is stored in the external flash memory because there is not enough space in on-chip memory. However, this external flash memory is much slower than the on-chip SRAM. For better performance, the access to the external memory should be made only when it is necessary. Caching frequently accessed data can help reducing flash memory accesses.

Another consideration is how much functionality is to be processed in sensor nodes. More sophisticated algorithms could reduce the overall programming time by reducing the network traffic, but at the cost of higher complexity computation and memory accesses.

Finally, the design should be simple so that it can be understood and diagnosed without difficulty.

3.1 Design: Fixed Block Comparison

As a starting point, we can design an incremental network programming by extending XNP. This consists of two main parts: (1) difference generation and code delivery, (2) storage organization and image rebuild.

Difference Generation

To generate the program difference, the host program compares each fixed sized block of the new program image with the corresponding block of the previous image. We set the block size as the page size of the external flash memory (256 bytes). The host program sends the difference as messages while it compares the two program versions. If the two corresponding blocks match, the host program sends a `CMD_COPY_BLOCK` message. The message makes the network programming module in the sensor node copy the block of the previous image to the current image. When the two blocks don't match, the host program falls back to the normal download; it sends a number of `CMD_DOWNLOADING` messages for the SREC records of the block (Figure 3.1).

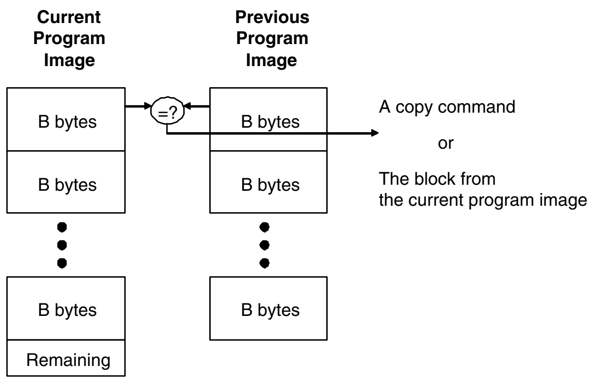


Figure 3.1: Generating difference with Fixed Block Comparison

The idea is that we can reduce the number of message transmission by sending a `CMD_COPY_BLOCK` message instead of multiple `CMD_DOWNLOADING` messages when most of the blocks are the same between the two program images.

Operations

Table 3.1 shows the message types used for the incremental network programming.

Message ID	Value	Description
`CMD_START_DOWNLOAD`	1	start network programming in normal mode
`CMD_DOWNLOADING`	2	deposit an SREC record
`CMD_QUERY_COMPLETE`	3	signals that it received all the capsules
`CMD_DOWNLOAD_STATUS`	4	request/response with download status
`CMD_DOWNLOAD_COMPLETE`	8	end of SREC record download
`CMD_ISP_EXEC`	5	execute the boot loader
`CMD_GET_PIDSTATUS`	7	Get Program ID
`CMD_GET_CIDMISSING`	6	Retransmission message from the host
`CMD_REQ_CIDMISSING`	6	Request retransmission for a missing cap
`CMD_START_DOWNLOAD_INCR`	10	start network programming incrementally
`CMD_COPY_BLOCK`	11	copy SREC records from previous to current
`CMD_GET_CURRENT_LINE`	22	Read the current SREC record
`CMD_GET_PREV_LINE`	23	Read the previous SREC record
`CMD_REPLY_LINE`	24	Reply to SREC record request

Table 3.1: Message types for incremental network programming

Based on XNP messages, we made the following extensions for the incremental network programming (Figure 3.2). The entire message format is shown in Appendix A.

- Start Download: `CMD_START_DOWNLOAD_INCR` message notifies the beginning of network programming in incremental mode. This message specifies not just the program ID of the current program but also the program ID of the previous program to ensure that the sensor node has the same program image as the host program.

- Download: Two operations `CMD_DOWNLOADING` and `CMD_COPY_BLOCK` are used to transmit the program image difference.

- Query and Reboot: The format of query, reply and reboot messages is the same as XNP messages.

- Debugging Messages: `CMD_GET_CURRENT_LINE` and `CMD_GET_PREV_LINE` messages request the SREC record at the specified line. In response, the sensor node sends `CMD_REPLY_LINE` message.

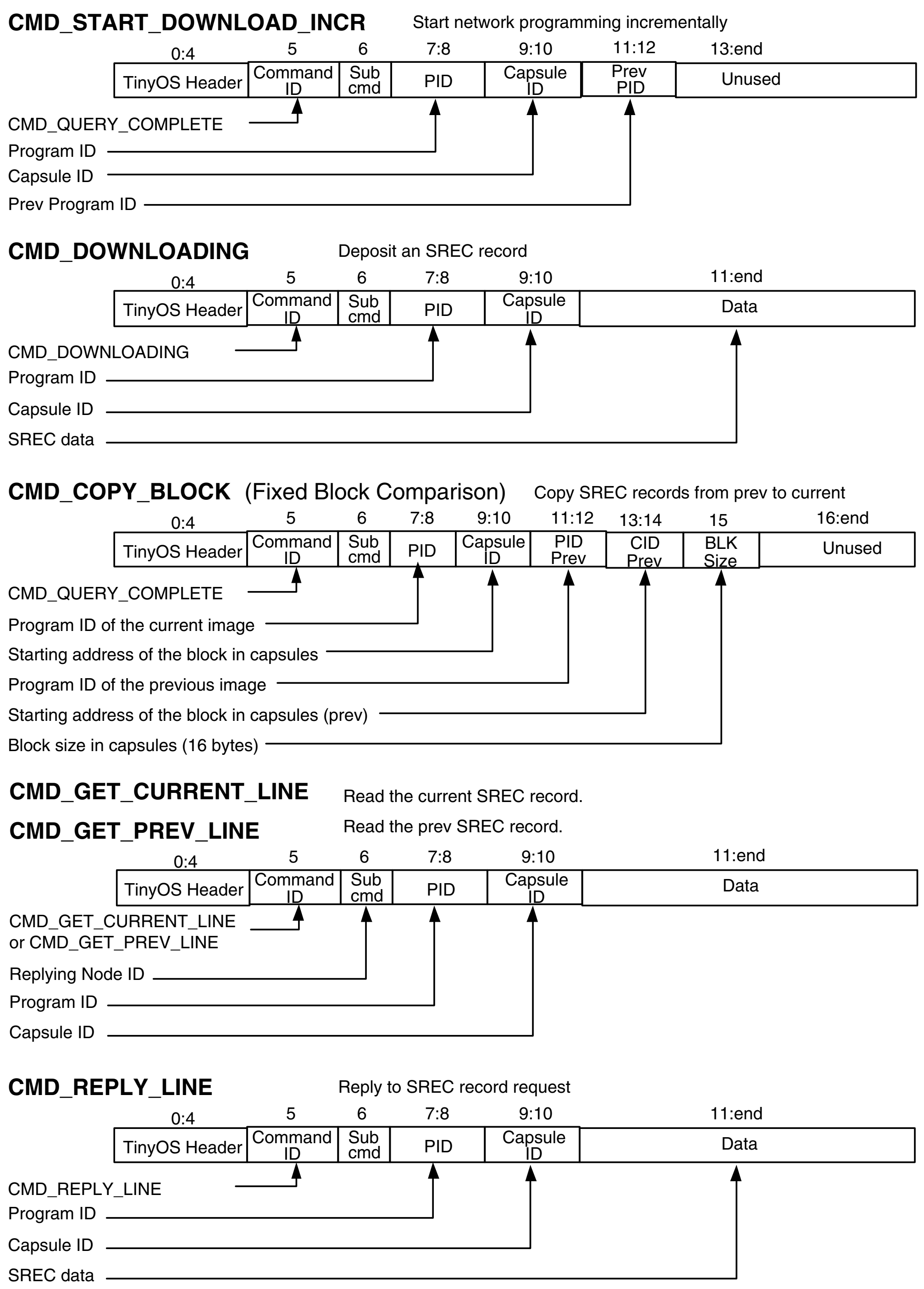


Figure 3.2: Protocol extension for incremental network programming

Storage Organization

XNP stores the program image in a contiguous memory chunk in the external flash memory. Fixed Block Comparison scheme extends this by allocating two memory chunks, one for the previous program image and the other for the scratch space where the current image will be built (Figure 3.3).

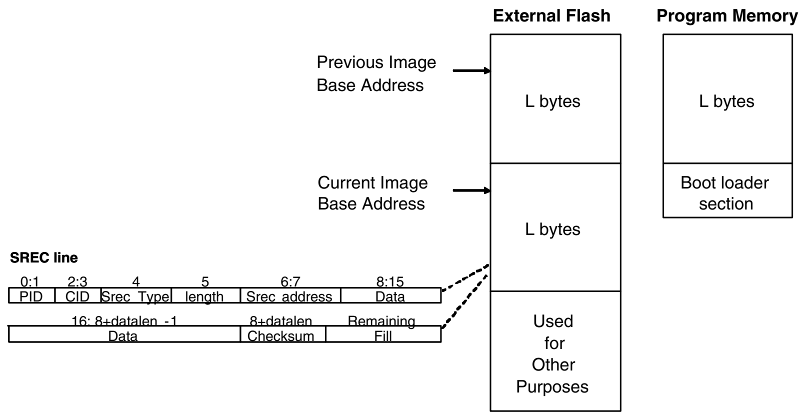


Figure 3.3: Memory allocation for Fixed Block Comparison

The two memory chunks have the same structure and they are swapped once the newly built program is loaded to the program memory. The current program image is now considered as the previous image and the space for the previous image is available for the next version of program image. For the two memory chunks, two base address variables are maintained in the flash memory. By changing the address values in these variables, the two memory chunks can be swapped.

This memory organization has an advantage that it provides the same view of the memory as XNP and minimizes the effort of rewriting the boot loader code. The boot loader code of XNP reads the program code assuming that it is located at a fixed location in the external flash memory. We modified the boot loader so that it reads the program code from the base address passed by interprocess call argument. Thus, the boot loader can read the program code from any memory chunk depending on the base address value passed by network program module.

However, this scheme does not use the external flash memory space economically. It allocates 256 KB of space regardless of the program code size (128 KB of space both for the current and the previous image). This accounts for 50% of the flash memory space of MICA2 motes and leaves less space for data logging.

Image Rebuild

The program image is built in a straightforward way. The network programming module of the sensor node builds the program image by writing the SREC records based on a list of download and copy messages (Figure 3.4).

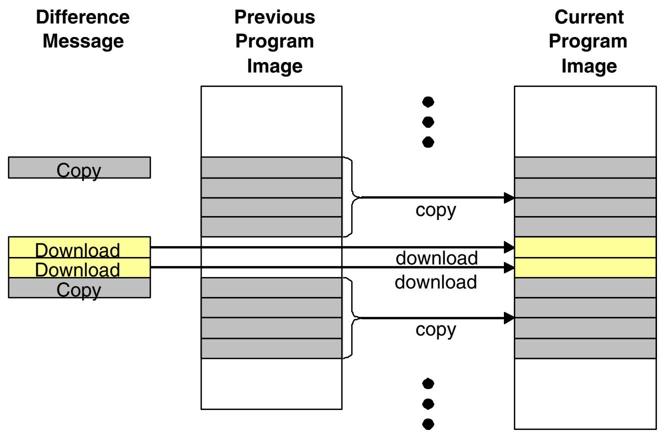


Figure 3.4: Program image rebuild with Fixed Block Comparison

The download message makes the sensor node deposit the data bytes in the message into the program image. The format of a download message is the same as an XNP download message. Capsule ID field specifies the location (line number) in the current program image and the data represents the data bytes to be written.

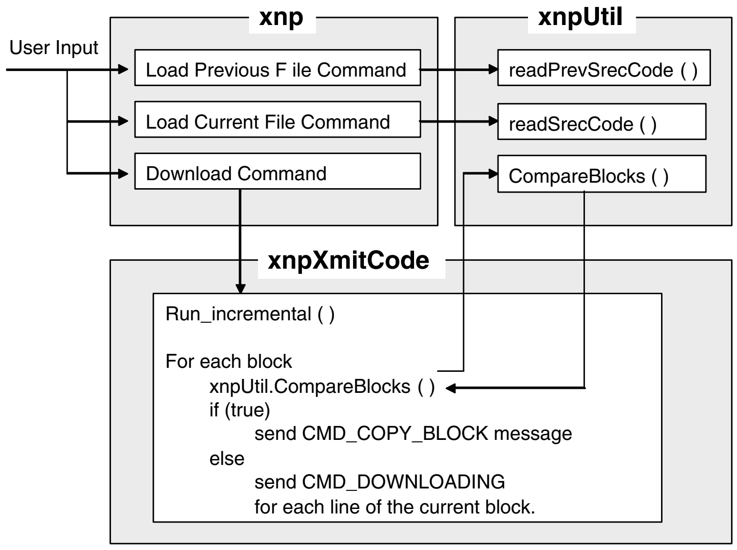
The copy message is introduced for the incremental network programming and it makes the sensor node copy the SREC lines of a block in the previous program image to the current program image. The capsule ID field specifies the location of the first SREC record to be copied and the block size field specifies the number of SREC records to be copied.

3.2 Implementation

Difference Generation and Code Delivery

The host program, which is in charge of program image loading, difference generation and code delivery, is composed of the following classes:

- xnp: GUI, main module
- xnpUtil: loads the program image, generates the difference and provides utility functions
- xnpQry: processes the query and retransmission
- xnpXmitCode: processes code delivery
- xnpMsg: defines the message structure
- MoteMsgIF: abstracts the interface to serial forwarder



```
Run_incremental ( )

For each block
      xnpUtil.CompareBlocks ( )
      if (true)
            send CMD_COPY_BLOCK message
      else
            send CMD_DOWNLOADING
            for each line of the current block.
```

Figure 3.5: Difference generation in the host program

If the user selects the download command after loading the current and the previous program images, `xnp` class spawns `xnpXmitCode` class. `xnpXmitCode` compares each pair of blocks in the current and previous images by calling `xnpUtil.CompareBlocks`. Depending on the result, it sends a copy message (`CMD_COPY_BLOCK`) or sends a download message (`CMD_DOWNLOADING`) for each line of the current block. Figure 3.5 illustrates this process.

Handling the Message

The network programming module for a sensor node is composed of the following components: XnpM.nc (implementation), XnpC.nc (configuration), Xnp.nc (interface), Xnp.h, XnpConst.h (constant definition). The implementation module has an event driven structure (Figure 3.6). When a XNP message arrives, `ReceiveMsg.receive()` sets the next state variable (`cNextState`) as the appropriate value and post `NPX_STATEMACHINE()` task. This message loop structure readily processes an incoming message without interrupting the message currently being processed. Tables B.1, B.2, B.4 show how an incoming message is processed.

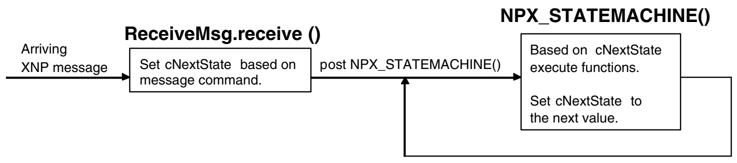

NPX_STATEMACHINE()
ReceiveMsg.receive ()
Arriving XNP message
Set cNextState based on message command.
post NPX_STATEMACHINE()
Based on cNextState execute functions.
Set cNextState to the next value.

Figure 3.6: Network programming module message handling

One of the difficult parts was handling split phase operations like external flash reads and writes. To read an SREC record from the external flash, `EEPROMRead.read()` is called. But this function returns before actually reading the record. The event handler `EEPROMRead.readDone()` is called when the record is actually read. And we specify the next state in the event handler. This makes us use multiple intermediate states to process an incoming message. Table B.2 and B.4 show which states were used to handle each message type.

To estimate the cost of message handling, we counted the source code lines for the two most important messages, `CMD_DOWNLOADING`, `CMD_COPY_BLOCK`. The number of lines are 136 and 153 respectively. Table B.5 shows the cost at each step of the message loop.

Calling the boot loader

XnpM builds the new program image based on the previous version and the difference. In order to transfer the new image to the program memory, we modified XnpM module and the boot loader.

The part of XnpM code that executes the boot loader is shown in the following. `wEEProgStart` is passed as the starting address of the new program image in the external flash memory.

```
task void NPX_ISP()
{
  ...
  wPID = ~wProgramID;  //inverted prog id
  __asm__ __volatile__ ("movw r20, %0" "\n\t"::"r" (wPID):"r20", "r21");
  wPID = wEEProgStart;
  __asm__ __volatile__ ("movw r22, %0" "\n\t"::"r" (wPID):"r22", "r23");
  wPID = wProgramID;   //the prog id
  __asm__ __volatile__ ("movw r24, %0" "\n\t"::"r" (wPID):"r24", "r25");

  //call bootloader - it may never return...
  __asm__ __volatile__ ("call 0x1F800" "\n\t"::);   //bootloader at 0xFC00
  ...
}
```

Here, `0x1F800` is the starting address of the boot loader. In Atmega128 microcontroller memory map, the boot loader can reside at one of the possible locations (Figure 3.2). We are using a boot loader of 4 KB in size.

The boot loader use the address passed as a parameter to access the new image as shown in the following code.

Word Address	Byte Address	Boot loader size
0xF000	0x1E000	8 KB
0xF800	0x1F000	4 KB
0xFC00	0x1F800	2 KB
0xFE00	0x1FC00	1 KB

Table 3.2: Possible boot loader locations

```
char _start(UINT16 wProgID, UINT16 wPageStart, UINT16 nwProgID, UINT8 param1)
{
  UINT16 EEPageA;
  UINT16 EEByteA;   //EEFlash Addresss
  ...
  EEPageA = wPageStart >> 4;
  EEByteA = 0;
  ...
  fEEStartRead( EEPageA, EEByteA);
  ...
}
```

3.3 Experiment Setup

To evaluate the performance of this design choice, we will count the number of block or packet transmissions of the test set. We considered the following three cases as a test scenario:

Case 1 (Changing Constants)

This is the case with the minimum amount of change. We modified the constant in `XnpBlink` that represents the blinking rate of the LED. `XnpBlink` is an application written for demonstrating the network programming (Appendix C). It accepts the network programming and blinks the red LED. The following code segment shows the modification to this program.

```
command result_t StdControl.start() {
  // Start a repeating timer that fires every 1000ms
  // The 1000 ms period can be changed with different value.
  return call Timer.start(TIMER_REPEAT, 1000);
}
```

Case 2 (Modifying Implementation File)

This is a more general case of program modification. We added a few lines of code to `XnpCount` program (Appendix C). `XnpCount` is a simple network programmable application. It counts a number, displays the number in its LEDs and broadcasts the number in radio packets. The following code segment shows the modification to this program.

```
event result_t Xnp.NPX_DOWNLOAD_DONE(uint16_t wProgramID,
               uint8_t bRet,uint16_t wEENofP){
  if (bRet == TRUE)
     call CntControl.start();
  else                        // this line can be deleted
     call CntControl.stop();  // this line can be deleted
  return SUCCESS;
}
```

Case 3 (Major Change)

In this case, we used two programs, `XnpCount` and `XnpBlink` as input to generate the difference. The difference is bigger than the first two cases. But, these two applications still share large part of the code in source level (Table 3.3).

	`XnpBlink`	`XnpCount`
# of source code lines for network programming modules	2049	2049
# of source code lines for application specific modules	157	198
# of SREC lines	1139	1166

Table 3.3: Code size of test applications

Case 4 (Modifying Configuration File – commenting out IntToLeds)

We commented a few lines in `XnpCount` program so that we do not use `IntToLeds` module. `IntToLeds` is a simple module that takes an integer input and displays it in LEDs of the sensor node. The following code segment shows the modification to this program.

```
configuration XnpCount {
}
implementation {
  components Main, Counter, /* IntToLeds,*/ IntToRfm, TimerC, XnpCountM,
             XnpC;

  // Main.StdControl -> IntToLeds.StdControl;
  // IntToLeds <- Counter.IntOutput;

}
```

Case 5 (Modifying Configuration File – commenting out IntToRfm)

We commented a few lines in `XnpCount` program so that we do not use `IntToRfm` module. `IntToRfm` takes an integer input and transmits it over the radio packet. Since commenting out `IntToRfm` makes the radio stack components not used, we expect bigger change in the program image than commenting `IntToLeds` module.

```
configuration XnpCount {
}
implementation {
  components Main, Counter, IntToLeds, /* IntToRfm,*/ TimerC, XnpCountM,
             XnpC;

  // Main.StdControl -> IntToRfm.StdControl;
  // Counter.IntOutput -> IntToRfm;

}
```

3.4 Results

To evaluate the performance of Fixed Block Comparison, we estimated the transmission time for each scenario. The host program calculates the estimated transmission time by counting how many download and copy messages it has sent. If it takes t_{down} to send a download message and t_{copy} to send a copy message, then the transmission time for Fixed Block Comparison , T, can be calculated as follows:

$$T = L_{down} \cdot t_{down} + N_{copy} \cdot t_{copy}$$

where L_{down} is the number of SREC lines sent by download messages and N_{copy} is the number of copy messages. As a baseline for comparison, we can also calculate the transmission time for the non-incremental delivery as follows:

$$T_{xnp} = L_{down} \cdot t_{down} + L_{copy} \cdot t_{down}$$

where L_{copy} is the number of SREC lines to be copied by a copy message. As for t_{down} and t_{copy}, we found right values after a number of trials. We set these as 120 ms and 300 ms respectively. Table 3.4 shows the parameters used for estimating the performance.

Parameter	**Description**
t_{down}	time to send a download message
t_{copy}	time to send a copy message
L_{down}	number of SREC lines sent by download message
L_{copy}	number of SREC lines transferred by copy message
N_{copy}	number of copy messages
T	transmission time of Fixed Block Comparison
T_{xnp}	transmission time of the non-incremental delivery

Table 3.4: Parameters for performance evaluation

Next we measured the transmission time by reading the system clock values. Table 3.5 shows the estimation and measurement data.

In case 1, the difference between the two program images is small. Most SREC

	Case 1	Case 2	Case 3	Case 4	Case 5
File size	48.9KB	50.1KB	50.1KB	49.7KB	49.6KB
Total SREC lines	1139	1167	1167	1156	1155
L_{down}	19	911	1135	1124	1123
L_{copy}	1120	256	32	32	32
N_{copy}	70	16	2	2	2
Estimation					
T	23280 ms	114120 ms	136800 ms	135480 ms	135360 ms
T_{xnp}	136680 ms	138720 ms	140040 ms	138720 ms	138600 ms
Speed-up (T_{xnp}/T)	5.87	1.22	1.02	1.02	1.02
Measurement					
T	25094 ms	124403 ms	149044 ms	147149 ms	146848 ms
T_{xnp}	149888 ms	152996 ms	152996 ms	150477 ms	150465 ms
Speed-up (T_{xnp}/T)	5.97	1.23	1.03	1.02	1.02

Table 3.5: Transmission time for each case

lines (1120 out of 1139) are transferred by copy messages and the speed up (T_{xnp}/T) is about 5.9.

In case 2, where we added a few lines in the source code, we find that less than a quarter of the SREC lines are transferred by copy messages (256 out of 1167) and the speed up is 1.2.

In case 3, only 32 out of 1167 lines are transferred by copy messages and the speed up is about 1.03. Even though `XnpBlink` and `XnpCount` share much in source code level, they have little sharing in binary code level. The main reason is that `XnpCount` uses the radio stack components while `XnpBlink` does not. The radio stack is one of the most important modules in TinyOS and it takes a number of source code lines.

In case 4 and 5, where we commented out `IntToLeds` and `IntToRfm` components in the configuration file `XnpCount.nc`, we find that only a small number of lines are transferred by copy messages and the speed up is very small (1.02 for each case).

Fixed block comparison was not so effective for the incremental network programming. It works well when the program structure doesn't change (case 1). But, the level of sharing was low when we added a few lines of code (case 2), which we think a more general case of program modification.

We want to see why we have such a small level of binary code sharing. Does the program code completely change after the source modification, or does the program code still have much similarity in byte levels ? For this purpose, we compared the program code in different levels: blocks (Fixed Block Comparison), SREC lines and bytes.

To compare program code in SREC lines, we used UNIX `diff` command. `diff` takes two ASCII files and describes how one file can be transformed to the other. To compare program code in byte level, we extracted the data bytes from an SREC file and stored each data byte in a line of the temporary file. We used UNIX `diff` to find the difference between the two byte list files.

Table 3.6 shows that case 2 and case 3 have much higher level of sharing in byte level than in block level. For case 2, most of binary code was similar in byte level (98.3%) while a small number of blocks were shared in block level (21.9 %). This implies that modifying the source code shifts the binary program code, but the program code bytes are still preserved. We can think of two ways to address this problem.

	Case 1	Case 2	Case 3
Blocks	97.2% (= $\frac{70}{72}$)	21.9% (= $\frac{16}{73}$)	2.7% (= $\frac{2}{73}$)
SREC lines	98.3% (= $\frac{1120}{1139}$)	40.8% (= $\frac{476}{1167}$)	12.0% (= $\frac{140}{1167}$)
Bytes	100.0% (= $\frac{18185}{18190}$)	98.3% (= $\frac{18320}{18636}$)	90.5% (= $\frac{16866}{18636}$)

Table 3.6: Level of code sharing in blocks, lines and bytes

One approach is to place the shared code at a fixed location in the binary code with the help of the compiler. We can insert compiler directives and inline function calls. Then, the compiler recognizes the network programming module and determines its location in the topological order.

Another approach is to utilize the code sharing without modifying the code. As Table 3.6 suggests, much of the binary code is shared in byte level. By comparing the two binary images at a variable size boundary like Rsync [11] and LBFS [12], we can find more chance of code sharing.

Chapter 4

Optimizing Difference Generation

Fixed Block Comparison, our first design choice for the incremental network programming, was not effective in reducing the data transmission traffic. It worked well only when the modified program image has the same structure has the previous program image. When additional lines are inserted in the source code, the program image is shifted and does not match the previous program image at the fixed sized block boundary.

In this section, we use the Rsync algorithm to generate the difference and rebuild the program image. The Rsync algorithm was originally made for efficient binary data update in a low bandwidth computer network. We expect the Rsync algorithm to find more matching blocks than the fixed block comparison because it compares the program image block at an arbitrary position.

4.1 Design

Difference Generation

The host program generates the difference using the Rsync algorithm as in Figure 4.1.

(1). The Rsync algorithm calculates a checksum pair (checksum, hash) for each fixed sized block (e.g. B bytes) of the previous program image. And the checksum pair is inserted into a lookup table.

(2). Rsync reads the current program image and calculates the checksum for the B byte block at each byte. If it finds a matching checksum in the lookup table, Rsync calculates the hash for the block and compares it with the corresponding entry in the table. If the hash also matches, then the block is considered a matching block.

(3). Rsync moves to the next byte for comparison if the block doesn't have a matching checksum or a hash. A region of bytes that doesn't have any matching blocks is tagged as non-matching block and needs to be sent explicitly for rebuild.

Figure 4.1 illustrates how the Rsync algorithm captures a matching block. Suppose there is a shift by a modification operation in the middle of the program image. Rsync forms a B byte window and calculates the hash for it. If the modified bytes are different from any blocks in the previous program image, then the hash of the modified bytes doesn't match any hash table entry with very high probability. Rsync moves the window one byte at a time and calculates the checksum for any possible match. It doesn't match until Rsync starts to read unmodified blocks. At this moment, Rsync has found a matching block.

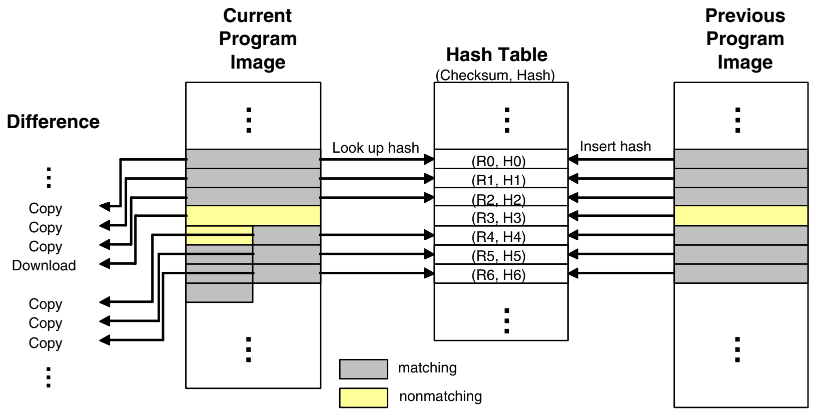


Figure 4.1: Rsync difference generation

Program Code Storage and Rebuild

As with the case of fixed block comparison, we maintain two memory chunks in a sensor node to build the program image from the previous program image and the difference. The difference consists of a list of matching and non-matching blocks.

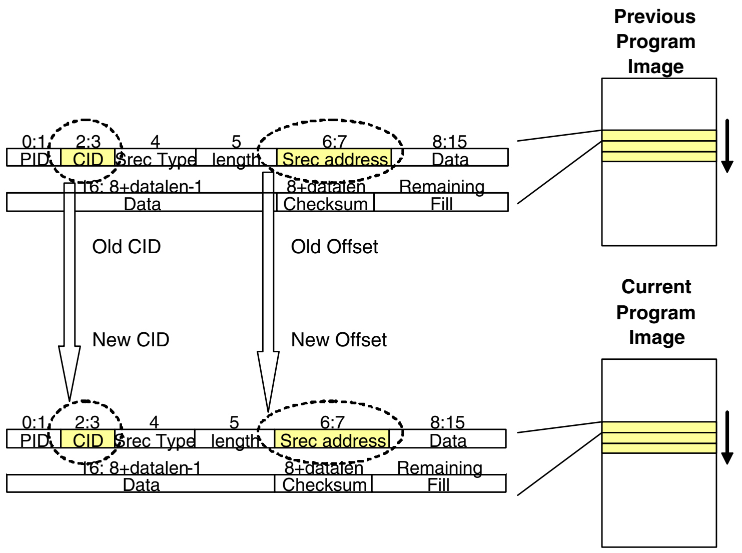


Figure 4.2: Copying a matching block

The host program sends a `CMD_COPY_BLOCK` message for each matching block in the difference. After hearing the message, the sensor node copies the block in the previous image to the current image. The block size of a copy message is a multiple of SREC line and the sensor node copies each SREC line iteratively. Since the block in the previous image can be mapped to any location in the current image, the offset address field of the SREC record needs be modified (Figure 4.2).

For each non-matching block in the difference, the host program sends one or multiple download (`CMD_DOWNLOADING`) messages. When a non-matching block is bigger than a single SREC record (16 bytes), the block is divided into multiple fragments and each fragment is sent in a download message. The data bytes of a download message can be

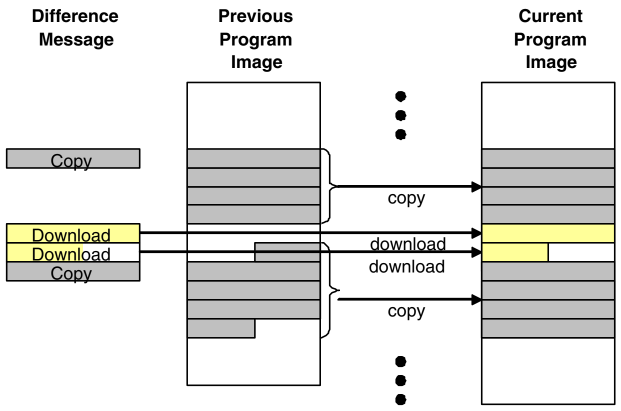


Figure 4.3: Downloading a non-matching block

shorter than a full SREC record if the non-matching block is not a multiple of 16 bytes. The host program does not fill the remaining bytes with the following data or copy messages. This is to avoid extra flash memory accesses although the resulting program image can have different layout with the original program image (Figure 4.3).

Unlike fixed block comparison, we use the base and the current program version to generate the program code incrementally. If we rebuild the current program image by comparing the last version and the current version, the host program and the sensor node may have different code and this can lead to incorrect program build. Instead, we compare the base and the current program version. This ensures that the sensor node reads the same data bytes as the host program.

Operations

We modified the format of `CMD_COPY_BLOCK` to specify the starting byte address of each copy block (Figure 4.4). When the Rsync algorithm generates the difference, the starting byte address of each block may not be a multiple of the SREC record size. We need to specify the starting byte address as well as the CID to correctly copy SREC records.

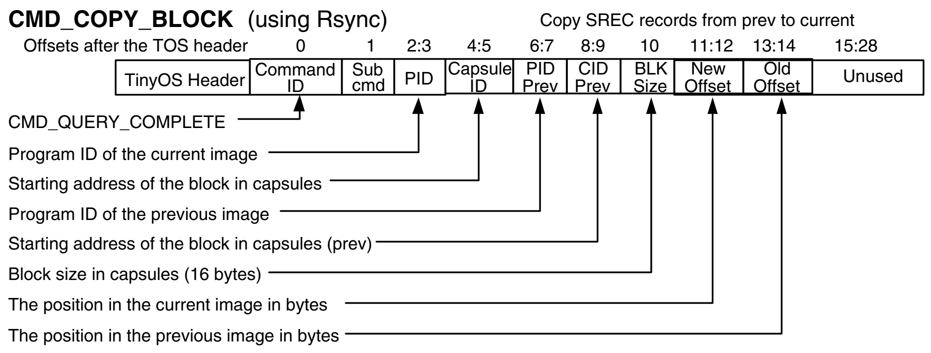

CMD_COPY_BLOCK (using Rsync)
Copy SREC records from prev to current
Offsets after the TOS header
0
1
2:3
4:5
6:7
8:9
10
11:12
13:14
15:28
TinyOS Header
Command ID
Sub cmd
PID
Capsule ID
PID Prev
CID Prev
BLK Size
New Offset
Old Offset
Unused
CMD_QUERY_COMPLETE
Program ID of the current image
Starting address of the block in capsules
Program ID of the previous image
Starting address of the block in capsules (prev)
Block size in capsules (16 bytes)
The position in the current image in bytes
The position in the previous image in bytes

Figure 4.4: CMD_COPY_BLOCK packet format modified for the Rsync algorithm

4.2 Implementation

Difference Generation

We used Jarsync [13] for the Rsync algorithm implementation. The host program calls the following methods to generate the difference: `Rdiff.makeSignatures()` and `Rdiff.makeDeltas()`. `makeSignatures()` calculates the checksum pair for each block in the image file and returns a list of checksum pairs. `makeDeltas()` compares the two image files and returns the difference as a list of matching blocks and unmatched blocks. Since these Jarsync methods assume a flat data file as input, the host program extracts only the data bytes from the SREC program image file and stores them in a temporary file before it calls the Jarsync module.

The difference returned by `makeDeltas()` needs postprocessing. The data bytes of an unmatched block can be an arbitrary size, but a download message can contain only up to 16 bytes. The host program divides an unmatched block into multiple blocks so that the data bytes of each block can fit in an SREC record. List entries for matching blocks are also postprocessed. Two matching blocks at the consecutive locations are merged into a bigger block and this reduces the number of message transmissions.

Program Code Storage and Rebuid

The rebuilt program can be different from the original file due to the missing packets. If the host program sends a query for the missing record (`CMD_GET_CIDMISSING`), the sensor node scans the current program section of the external flash memory. Each record contains program ID (PID) and the capsule ID (CID, sequence number) fields. The PID should match the PID advertised at the start of the incremental network programming (`CMD_START_DOWNLOAD_INCR`). The CID field should match the line number where the record is written to. If either PID or CID does not match, then the sensor node considers this a missing record and requests the retransmission of the SREC record. The host finds the missing record and sends it back. Then, the sensor node can fill the hole.

When the sensor node requests the retransmission of a missing SREC record, it specifies the missing record by CID field. Since the rebuilt program image can have different layout from the original program file, just reading the specified record in the original program file does not return the correct data. To address this issue, the host program rebuilds the new program that has the same layout with the program image to be built in a sensor node. And the host program reads SREC records of this image for retransmission requests.

Code Complexity

To estimate the complexity of our implementation, we counted the source code lines in `XnpM.nc` file. A `CMD_DOWNLOADING` message costs 136 lines and a `CMD_COPY_BLOCK` message (for Rsync) costs 153 lines. The details are shown in Table B.5. These numbers are comparable to those of other TinyOS modules. Sending and receiving radio packets are handled in several modules and `CC1000RadioIntM.nc` is core module. A send operation takes 112 lines and a receive operation takes 88 lines in this module. As another example, we analyzed `ADCM.nc` module which handles the reading the data from an ADC channel. It takes 35 lines to get a byte data with `ADCM.nc`. Table 4.1 summarizes this.

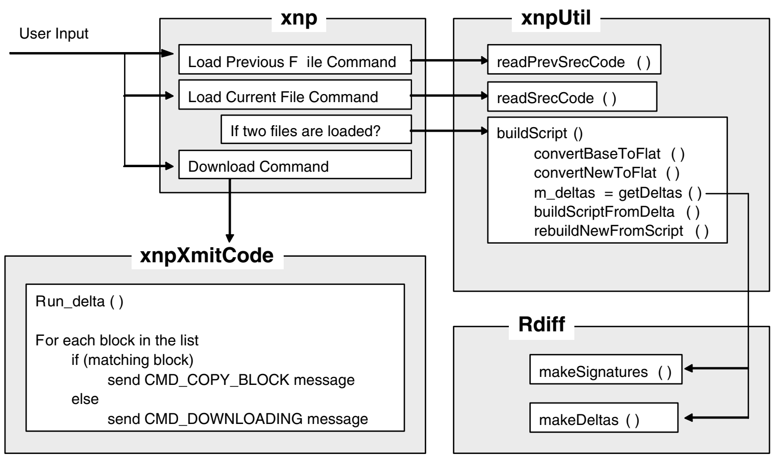


Figure 4.5: Host program with Rsync implementation

Incremental Network Programming		**Radio Stack MAC**		**ADC Operation**
Download	Copy (Rsync)	Send	Receive	Get and DataReady
136	153	112	88	35

Table 4.1: Complexity of incremental network programming

4.3 Results

To evaluate the performance of the incremental network programming with the Rsync algorithm, we estimated and measured the transmission time for the three cases: (1) changing a constant in `XnpBlink`, (2) adding a few lines in `XnpCount` and (3) transforming `XnpBlink` to `XnpCount`. Table 4.2 shows the results.

In case 1, most SREC records (1116 lines out of 1120) were transferred and the speed up over the non-incremental delivery was 6.25 (measurement). This is almost the same as the speed up of Fixed Block Comparison (Case 1 in Figure 4.6).

In case 2, 954 lines out of 1154 lines were transferrred by copy messages and the speed up over the non-incremental delivery was 2.44 (measurement). Whereas Fixed Block Comparison has speed up of 1.2 (Case 2 in Figure 4.6). The improved speed up is caused

	Case 1	Case 2	Case 3	Case 4	Case 5
File size	48.2KB	49.4KB	49.4KB	48.9KB	48.9 KB
Total SREC lines	1120	1154	1156	1140	1147
L_{down}	4	200	888	326	871
L_{copy}	1116	954	278	814	276
N_{copy}	72	104	85	107	83
Estimation					
T	22080 ms	55200 ms	132060 ms	71220 ms	129420 ms
T_{xnp}	134400 ms	138480 ms	139920 ms	136800 ms	137640 ms
Speed-up (T_{xnp}/T)	6.09	2.51	1.06	1.92	1.06
Measurement					
T	23812 ms	61015 ms	142607 ms	77090	140314
T_{xnp}	148823 ms	148889 ms	148889 ms	148172	148016
Speed-up (T_{xnp}/T)	6.25	2.44	1.04	1.92	1.05

Table 4.2: Transmission time with the Rsync algorithm

by the efficient difference generation of the Rsync algorithm.

In case 3, the level of sharing was much smaller and the speed up was 1.04 (measurement). We have some number of copy messages (85 messages), but they cover only a small number of blocks and are not so helpful in reducing the programming time.

In case 4, 814 lines out of 1140 lines were transferrred by copy messages and the speed up over the non-incremental delivery was 1.92 (measurement). In contrast, the speed up with Fixed Block Comparison was almost negligible (1.02).

In case 5, 276 lines out of 1140 lines were transferrred by copy messages and the speed up over the non-incremental delivery was quite small – 1.06 (measurement). Both case 4 and case 5 commented a few lines in the configuration file. But, in case 5, commenting out `IntToRfm` component made the radio stack not used and this changed the layout of the program image file a lot.

In summary, using the Rsync algorithm achieves the speed up of 6 for changing the constant and 2.4 for adding a few source code lines. These numbers are bigger than those of Fixed Block Comparison, but using the Rsync algorithm is not still effective with the major code change.

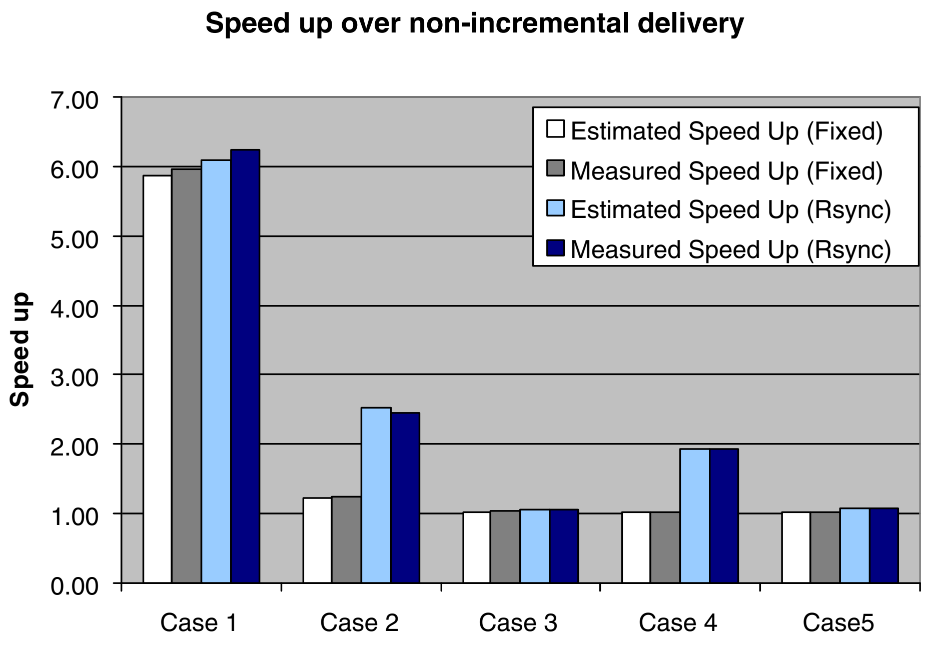


Figure 4.6: Comparison of programming time

As for the results in Table 4.2, we have some comments.

First, we can ask why 4 SREC lines were transmitted as download messages in case 1 when we changed only a constant in the source file. One of the reason is that the network programming module includes a timestamp value that is given at each compile time. This ensures that each program image is different each time we compile the program. Another reason is that the previous SREC file is not aligned in the SREC record boundary at the end of the file. When we convert the SREC file to a flat file for Rsync, the layout changes.

Another question is that why we sent 72 copy messages even though we could send fewer messages. In our design, the sensor node copies the program image blocks after hearing a copy message. To bound the execution time, we made each copy message handle up to 16 SREC lines (256 bytes).

Chapter 5

Optimizing Difference Delivery

Compared to Fixed Block Comparison, the Rsync algorithm achieves the shorter programming time by efficiently finding the shared blocks between the two binary code files. However, we can find something to improve:

First, the network programming module transfers only a limited number of SREC records for each copy message. This is to bound the running time of a copy message so that the network programming module finishes processing a copy request before it receives another request.

Second, the network programming module interprets a copy request right after it receives the request without saving the request. In case there is a missing command, the network programming module has to check the rebuilt program image because it hasn't stored the script commands. Since the network programming module does not know whether a missing hole is caused by a missing copy message or a number of download messages, it sends a retransmission requests for each missing record in the current program image. This will take more time than retransmitting only the missing command.

Thus, we propose extending the implementation of Chapter 4 as follows:

(1). The sensor node receives all the commands for the script.

(2). The sensor node checks any missing records for the script.

(3). The sensor node starts to decode script records in response to the script decode message.

5.1 Design

Operations

Since the script commands are stored in the storage space of the sensor node, we modified `CMD_DOWNLOADING` message to send script messages. This has an advantage that we can reuse most of the code for handling normal data records to process the script commands.

Figure 5.1 shows the packet format for the data download message. It is almost the same as the format for the normal data record download message except the script CID and new CID fields. Script CID field is the sequence number of the command within the script and the new CID field is the location where the data record embedded in the command will be copied for building the program image.

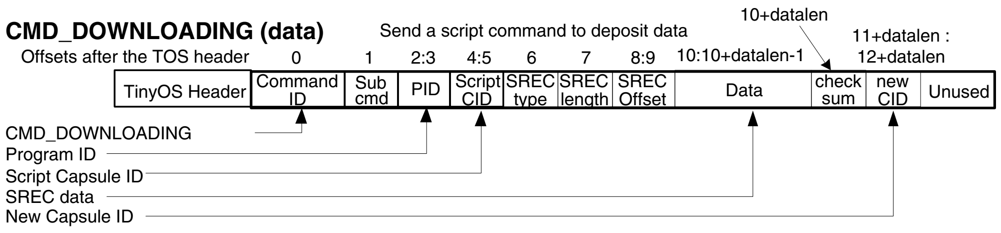


Figure 5.1: Message format for CMD_DOWNLOADING (data)

Figure 5.2 shows the packet format for a copy command. This command is also stored in a similar way as a normal data record. A copy command has the SREC type field. This is for the Motorola SREC type and only several values are allowed by the specification (0,1,2,3,5,7,8 and 9). We extended the meaning of this field so that the value 10 represents a copy record. This allows us to store a copy command in the same way as other data records, but still interprets the copy command correctly.

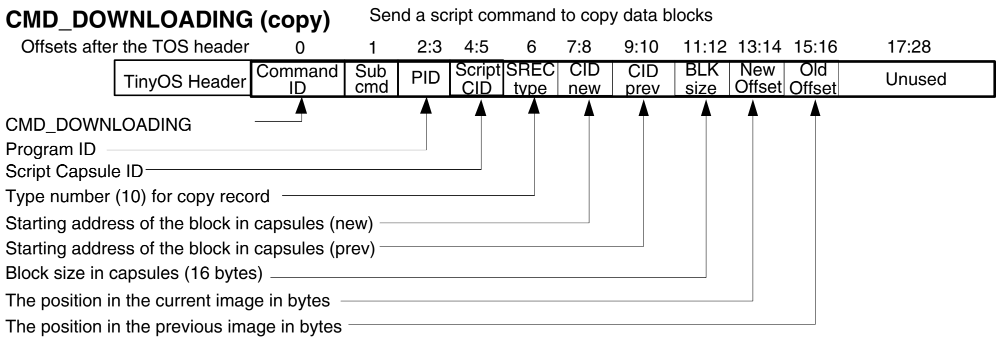

CMD_DOWNLOADING (copy)
Send a script command to copy data blocks
Offsets after the TOS header
0
1
2:3
4:5
6
7:8
9:10
11:12
13:14
15:16
17:28
TinyOS Header
Command ID
Sub cmd
PID
Script CID
SREC type
CID new
CID prev
BLK size
New Offset
Old Offset
Unused
CMD_DOWNLOADING
Program ID
Script Capsule ID
Type number (10) for copy record
Starting address of the block in capsules (new)
Starting address of the block in capsules (prev)
Block size in capsules (16 bytes)
The position in the current image in bytes
The position in the previous image in bytes

Figure 5.2: Message format for CMD_DOWNLOADING (copy)

Figure 5.3 shows the packet format for the decode command. Decode message makes the network programming module start decoding downloaded script commands.

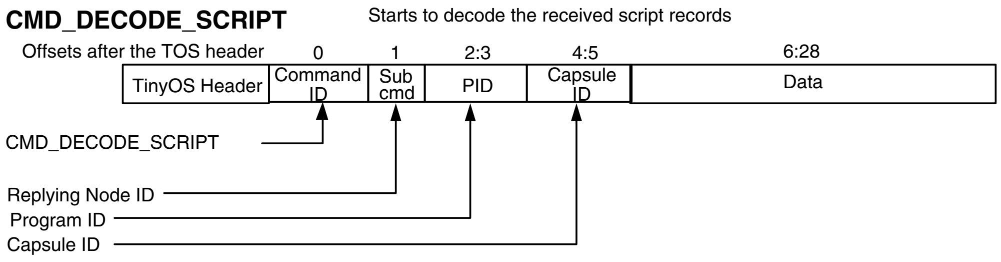


Figure 5.3: Message format for CMD_DECODE_SCRIPT

Storage Organization and Program Rebuild

As the storage space for the script commands, we need to choose among RAM, internal flash memory and external flash memory. RAM would be better than the others for its fast access time. However, the size of a script can be as large as a list of download messages in worst case. Since the largest program size is 128 KB, it may not fit into RAM

(4 KB) or the internal flash memory (4 KB) when the program size is big. Thus, the script should be stored in the external flash memory.

We divided the external flash memory into three sections: the previous program image, the current program image and the script sections.

At first, the host program sends the script as `CMD\_DOWNLOADING` messages. The sensor node stores these messages in the script section if it is in the incremental network programming state. This is shown in Figure ??.

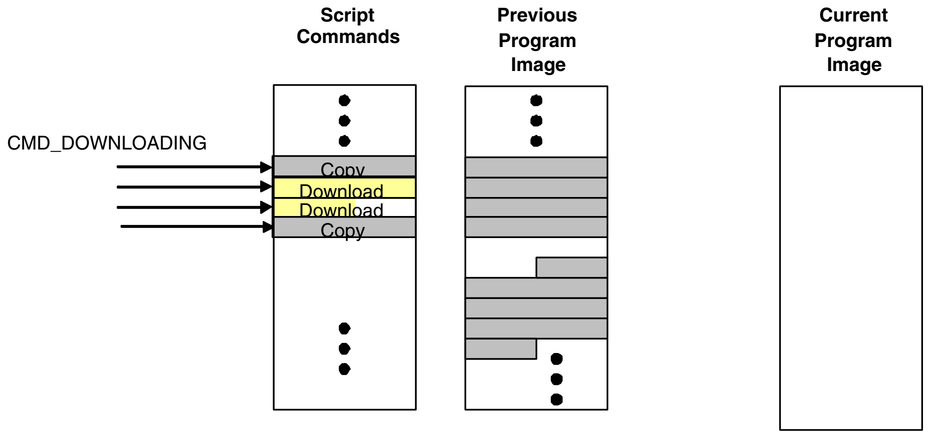


Figure 5.4: Receiving script commands

When the host program queries any missing script commands, the sensor node scans the script section. When the difference between the two program versions is small, the traversal of the script section can finish quickly. If the sensor node finds any missing record, then it requests the retransmission of the record. Then, the host program sends the record again.

After receiving the decode command from the host program, the sensor node starts rebuilding the program code. This is shown in Figure 5.5.

A download command is copied from the script section to the current program image section after the CID field is modified to the new CID value. As for a copy command,

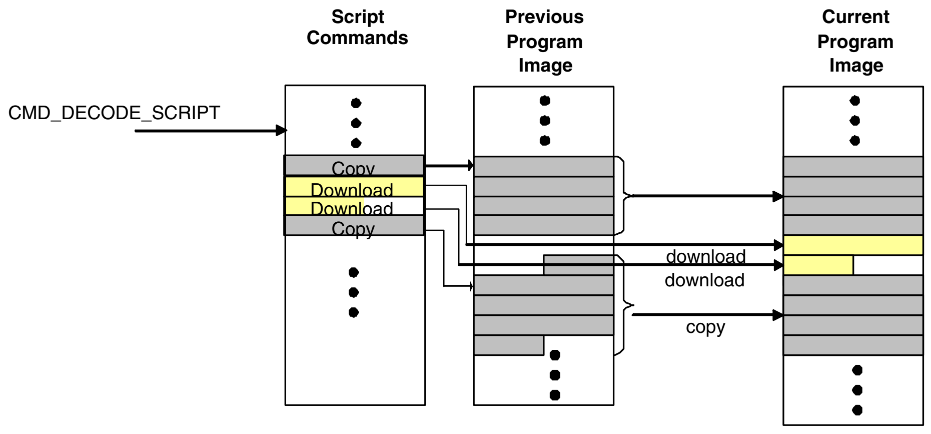


Figure 5.5: Decoding script commands

the sensor node starts copying SREC records from the previous program image to the current program image. An SREC record from the previous section is copied to the current program section after the CID and the byte offset fields are modified for the new values.

5.2 Results

Since a sensor node does not rebuild the program image until it receives all the script commands, we modified the metrics for the evaluation. We measured the transmission time and the decode time for the three cases.

The host program saves the time stamp value when it sends a decode command and gets the next time stamp value when it receives the reply from the sensor node. The decode time is calculated as the difference of the two time stamp values. Table 5.1 shows the results.

For case 1, only 7 script messages were transmitted and this made the transmission time very small. The sum of transmission time and the decode time is 16015 ms while the non-incremental delivery took 154043 ms. This gives the speed up of 9.10

For case 2, more script lines were transmitted (337 script messages for the 1167

	Case 1	Case 2	Case 3	Case 4	Case 5
File size	48.9KB	50.1KB	50.1KB	49.7KB	49.7KB
Total SREC lines	1139	1167	1167	1156	1156
Number of commands	7	337	996	419	964
Measurement					
T	922 ms	45843 ms	130653 ms	54544	125577
T_{decode}	16015 ms	16687 ms	16874 ms	16765	16796
T_{xnp}	154043 ms	158481 ms	158481 ms	150654	150525
Speed-up (T_{xnp}/T)	9.10	2.53	1.07	2.11	1.06

Table 5.1: Transmission time with the Rsync algorithm and script decode

line program code) and the speed up over the non-incremental delivery is 2.53.

For case 3, we sent even larger number of script messages (996 messages for 1167 line program code) and the speed up was 1.07.

When we modified the configuration file, we had the similar results with the Chapter 4. For case 4, 419 script messages for the 1156 line program code and the speed up over the non-incremental delivery is 2.11. For case 5,most of the SREC records were transmitted as download script commands (964 out of 1156) and the speed up was 1.06.

Figure 5.6 and Table 5.2 show the results of the three incremental network programming implementations: Fixed Block Comparison, Rsync and Rsync with split decode. We can find that splitting the script transmission and the program rebuild improves the overall programming time. When the source code is modified at minimum, the implementation with Rsync and split decode saved programming time by sending fewer script messages even though it has to decode the script messages. When a small number of source code lines were added, the programming time was a little better than the implementation that just uses the Rsync algorithm. For the major program change, it didn't achieve the speed up, but it was still as good as the non-incremental delivery.

We can comment on case 3. Even though we used the Rsync algorithm and split decode, the speed up over the non-incremental delivery was negligible. This is because the difference between the two program images cannot be described with a small number of insert, copy and skip operations.

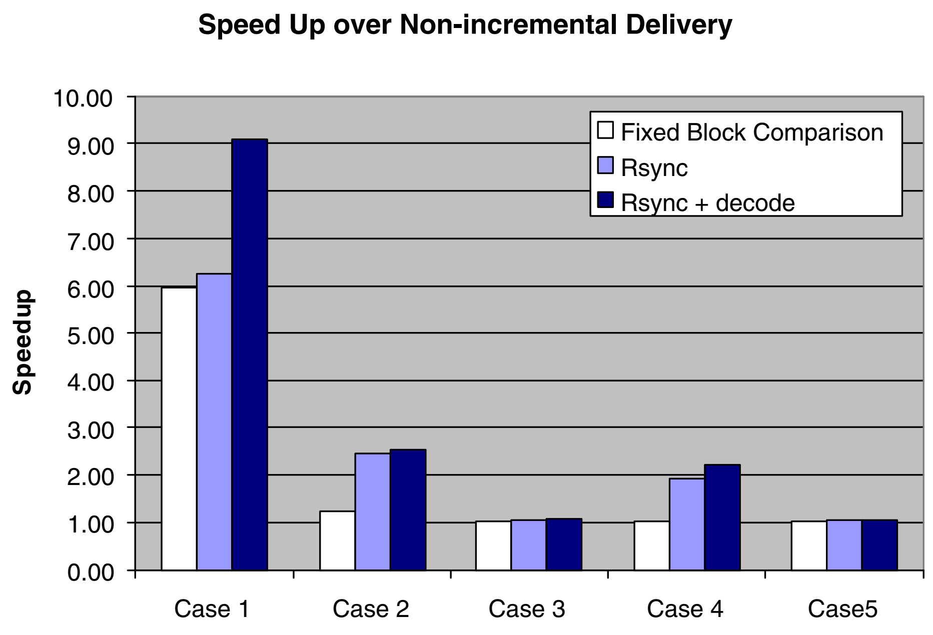


Figure 5.6: Comparison of programming time

Fixed Block Comparison

	Case 1	Case 2	Case 3	Case 4	Case 5
File size	48.9KB	50.1KB	50.1KB	49.7KB	49.6KB
Total SREC lines	1139	1167	1167	1156	1155
L_{down}	19	911	1135	1124	1123
L_{copy}	1120	256	32	32	32
N_{copy}	70	16	2	2	2
T	23280 ms	114120 ms	136800 ms	135480 ms	135360 ms
T_{xnp}	136680 ms	138720 ms	140040 ms	138720 ms	138600 ms
Speed-up (T_{xnp}/T)	5.87	1.22	1.02	1.02	1.02

Rsync

	Case 1	Case 2	Case 3	Case 4	Case 5
File size	48.2KB	49.4KB	49.4KB	48.9KB	48.9 KB
Total SREC lines	1120	1154	1156	1140	1147
L_{down}	4	200	888	326	871
L_{copy}	1116	954	278	814	276
N_{copy}	72	104	85	107	83
T	22080 ms	55200 ms	132060 ms	71220 ms	129420 ms
T_{xnp}	134400 ms	138480 ms	139920 ms	136800 ms	137640 ms
Speed-up (T_{xnp}/T)	6.09	2.51	1.06	1.92	1.06

Rsync with split decode

	Case 1	Case 2	Case 3	Case 4	Case 5
File size	48.9KB	50.1KB	50.1KB	49.7KB	49.7KB
Total SREC lines	1139	1167	1167	1156	1156
Number of commands	7	337	996	419	964
T	922 ms	45843 ms	130653 ms	54544	125577
T_{decode}	16015 ms	16687 ms	16874 ms	16765	16796
T_{xnp}	154043 ms	158481 ms	158481 ms	150654	150525
Speed-up (T_{xnp}/T)	9.10	2.53	1.07	2.11	1.06

Table 5.2: Comparison of programming time

Chapter 6

Related Work

6.1 Wireless Sensor Network Programming

XNP [1], [2] is the network programming implementation for TinyOS that was introduced with 1.1 release version. XNP supports basic network programming - broadcasting the program code to multiple nodes in a single hop. It doesn't consider a large sensor network and incremental update.

MOAP [4] is a multihop network programming mechanism developed by Stathopoulos et al. Main contributions of MOAP is its code dissemination and buffer management. One of the challenges of multihop network programming is propagating program code over multiple sensor nodes without saturating the network. They used Ripple dissemination protocol to regulate the network traffic. Ripple protocol disseminates the program code packets to a selective number of nodes without flooding the network with packets. For buffer management, they used sliding window scheme. Sliding window scheme maintains a window of program code and allows lost packets within the window to be retransmitted. Sliding window takes small footprint so that packets can be processed efficiently in on-chip RAM. MOAP was tested in EmStar simulator and MICA2 motes.

Deluge [5] is a multihop network programming protocol developed by Hui and

Tolle. Deluge disseminates program code in an epidemic fashion to propagate program code while regulating the excess traffic. The epidemic algorithm works like this: Some nodes advertise the version of code they have. After hearing the advertisement, the nodes missing any of pages make requests for the missing pages. Then, sender nodes transmit the requested pages. In order to increase the transmission rate, Deluge used optimization techniques like adjusting packet transmit rate and spatial multiplexing. Unlike MOAP, Deluge uses a fixed sized page as a unit of buffer management and retransmission. Deluge was tested with TOSSIM simulator [15] and MICA2 motes.

Reijers et al [3] developed algorithms for updating binary image incrementally. In their algorithm, the host program generates “edit script” to describe the difference between the two program code and the sensor nodes build the program image after interpreting the edit script. The edit script consists of not only simple operations like copy and insert but also more complex operations (address repair and address patch) that modify the program code at the instruction level. This helps minimizing the edit script size. As an evaluation, this paper considers only the reduced script size on the host side. Since operations like address repair and address patch incur memory intensive EEPROM scanning, the experiments should have demonstrated the overall programming time in a sensor simulator or in a real sensor node.

Kapur et al [6], [7] implemented an incremental network programming based on the algorithm of Reijers et al [3]. Their implementation is composed of two parts: the diff encoder on the host side and the diff decoder on the sensor node side. The diff encoder generates the difference for the two versions of code at instruction level using copy, insert and repair operations. The difference script is delivered to the sensor node using MOAP [4] which was developed for reliable code dissemination. Then, the sensor node rebuilds the program code after decoding the downloaded script.

These two works on the incremental network programming minimized the script transmission at the cost of program modification at the instruction level. In contrast, the implementation in this thesis put less computational complexity on the sensor nodes. The difference generation, which is costly, is handled by the host program. The sensor nodes just write the data blocks based on the script commands and this can be applied to less powerful sensor nodes.

While the examples above disseminated the program code in native binary code, Maté [8] distributes the program code in virtual machine instructions which are packed in radio packets. While XNP transmits the binary code that contains both the network programming module and the application, Maté only transmits the application code. This allows Maté to distribute the code more quickly. One drawback of Maté is that it runs the program code only in virtual machine instructions and a regular sensor application needs to be converted to virtual machine instructions before execution.

Trickle [9] is an improvement over Maté. In Maté, each sensor node floods the network with packets to distribute the code and this can lead to network congestion and the algorithm can be used for a large sensor network. Trickle addresses this problem by using "polite gossip" policy. Each sensor node periodically broadcasts a code summary to its local neighbors and it stays quiet if it has recently heard a summary identical to its summary. The sensor node broadcast an update only when it hears from an older summary than its own.

6.2 Remote Code Update outside Wireless Sensor Community

Outside the sensor network community, there have been efforts to update program code incrementally. Emmerich et al [10] demonstrated updating XML code in an incremental fashion. Specifying the update in XML is easier than in binary image because XML is a structured markup language and it allows specifying the update without changing the structure of the rest of the code. In contrast, inserting or replacing code blocks in binary code affects the rest of the code.

The cases of synchronizing general form of unstructured files can be found with Rsync and LBFS. Rsync [11] is a mechanism to efficiently synchronize two files connected over a low-bandwidth, bidirectional link. To find matching blocks between the two files, we can divide the first file into fixed sized blocks of B bytes and calculate the hash for each block. Then, we scan the second file and form a B byte window at each byte. After that we compare the hash for the window with hash values of all the blocks in the first file. This does not work that well. If the hash is expensive to calculate, then finding the matching blocks will take long time. If the hash can be computed cheaply but with false match, we do not find the correct block. The key idea of Rsync is to use two levels of hashes, rolling checksum (fast hash) and hash (strong hash) to make the computation overhead manageable while finding the matching blocks with high probability. Rsync calculates the rolling checksum of the B byte window of the second file at each byte and computes the hash only when the rolling checksums of the two blocks match. Since the hash is computed only for the possible matches, the cost of calculating hash is managable and we can filter out the false match.

LBFS [12], another mechanism to synchronize two files in a low-bandwidth, bidirectional link, takes a slightly different approach. Rather than divides a file into fixed blocks, LBFS divides each file into a number of variable sized blocks, computes the hash over each block. To find matching blocks between the two files, LBFS just compares these hashes (SHA-1 hash). The key idea of LBFS is in dividing a file into variable sized blocks. LBFS scans a file and forms 48-byte window at each byte and calculates 13-bit fingerprint. If the fingerprint matches a specific pattern, then that position becomes the breakpoint of the block. This scheme has a property that modifying a block in a file does not change the hash values of the other blocks. When we are going to send a new version, we can just compare the hash values of the each variable blocks and sends only the non-matching blocks.

The patent filed by Metricom Inc. [14] describes a mechanism that disseminates the program code over the multihop network in an efficient way. When a node V has a new version of code, it tells its neighbors that a new version of code is available. On hearing the advertisement from V, one of V's neighbor, P, checks whether it has the newly advertised version. If it doesn't have the version, P requests V transmit the version of code. After that, V starts sending program code and finishes when it doesn't hear any requests. With this scheme, a sensor node can distribute the program code without causing much network traffic.

Chapter 7

Conclusion

The network programming is a way of programming wireless sensor nodes by sending the program code over the radio packets. By sending program code packets to multiple sensor nodes with a single transfer, the network programming saves the programming efforts for a large sensor network. The network programming implementation in the current TinyOS release provides the basic capability of the network programming – delivering the program code to the sensor nodes remotely. However, the network programming implementation is not optimized when part of the program code has changed. It transmits all the code bytes even though the new version of program code is different only in small amount.

We extended the network programming implementation so that it reduces the programming time by transmitting the incremental update rather than the whole program code. The host program generates the difference of the two program images using the Rsync algorithm and transmits the difference to the sensor nodes. Then, the sensor nodes decode the difference script and build the program image based on the previous program version and the difference script. We tested our incremental network programming implementation with some test applications. We have speed up of 9.1 for changing a constant and 2.1 to 2.5 for chaning a few lines of code in the source code.

For future work, we plan to incorporate multihop delivery like Deluge [5] and

MOAP [4] into our incremental network programming implementation. This will allow deploying incremental network programming over a large network. Our implementation reliably delivers the difference script for a single hop case by extending the message transmission of XNP. To support multihop incremental network programming, we can disseminate the difference script and the decode command using the available multihop delivery implementations.

We also plan to apply optimization techniques for TinyOS nesC compiler. We believe this will capture the source level sharing which is not handled by the Rsync algorithm.

The source for this thesis is publicly available in the following web site:
`http://www.cs.berkeley.edu/~jaein/master_thesis`

Bibliography

[1] Jaein Jeong, Sukun Kim and Alan Broad, "Network Reprogramming," TinyOS document, http://webs.cs.berkeley.edu/tos/tinyos-1.x/doc/NetworkReprogramming.pdf

[2] Crossbow Technology, "Mote In Network Programming User Reference," TinyOS document, http://webs.cs.berkeley.edu/tos/tinyos-1.x/doc/Xnp.pdf

[3] Niels Reijers and Koen Loangendoen, "Efficient Code Distribution in Wireless Sensor Networks," *WSNA '03*

[4] Thanos Stathopoulos, John Heidemann and Deborah Estrin, "A Remote Code Update Mechanism for Wireless Sensor Networks," *CENS Technical Report # 30,* http://lecs.cs.ucla.edu/~thanos/moap-TR.pdf

[5] Adam Chlipala, Jonathan Hui and Gilman Tolle, "Deluge: Data Dissemination in Multi-Hop Sensor Networks," *UC Berkeley CS294-1 Project Report, December 2003,* http://www.cs.berkeley.edu/~jwhui/research/projects/deluge/deluge_poster.ppt

[6] Rahul Kapur, Tom Yeh and Ujjwal Lahoti, "Differential Wireless Reprogramming of Sensor Networks," *UCLA CS213 Project Report, December 2003*

[7] Tom Yeh, Haru Yamamoto and Thanos Stathopolous, "Over-the-air Reprogramming of Wireless Sensor Nodes," *UCLA EE202A Project Report, December 2003,* http://www.cs.ucla.edu/~tomyeh/ee202a/project/EE202a_final_writeup.doc

[8] Philip Levis and David Culler, "Maté: A Tiny Virtual Machine for Sensor Networks," *ASPLOS Oct. 2002*

[9] Philip Levis, Neil Patel, Scott Shenker, and David Culler "Trickle: A Self-Regulating Algorithm for Code Propagation and Maintenance in Wireless Sensor Networks," In *Proceedings of the First USENIX/ACM Symposium on Networked Systems Design and Implementation (NSDI 2004).*

[10] Wolfgang Emmerich, Cecilia Mascolo and Anthony Finkelstein, "Implementing Incremental Code Migration with XML," *Proceedings of the 22nd International Conference on Software Engineering, 2000*

[11] Andrew Tridgell, "Efficient Algorithms for Sorting and Synchronization," *PhD thesis, Austrailian National University, 1999*

[12] A. Muthitacharoen, B. Chen, and D. Mazieres, "A low-bandwidth network file system," *Proc. 18th SOSP, pages 174–187, Oct. 2001*

[13] Casey Marshall, "Jarsync: a Java implementation of the rsync algorithm," http://jarsync.sourceforge.net/

[14] George H. Flammer, III, "Method for Distributing Program Code to Intelligent Nodes in a Wireless Mesh Data Communication Network," *US Patent No. 5,903,566, May 11, 1999*

[15] Phil Levis et al, "TOSSIM: A Simulator for TinyOS Networks," http://today.cs.berkeley.edu/tos/tinyos-1.x/doc/nido.pdf

[16] Atmel, "ATmega 128 Microcontroller Reference," http://www.atmel.com/dyn/resources/prod_documents/doc2467.pdf

Appendix A

Message Format

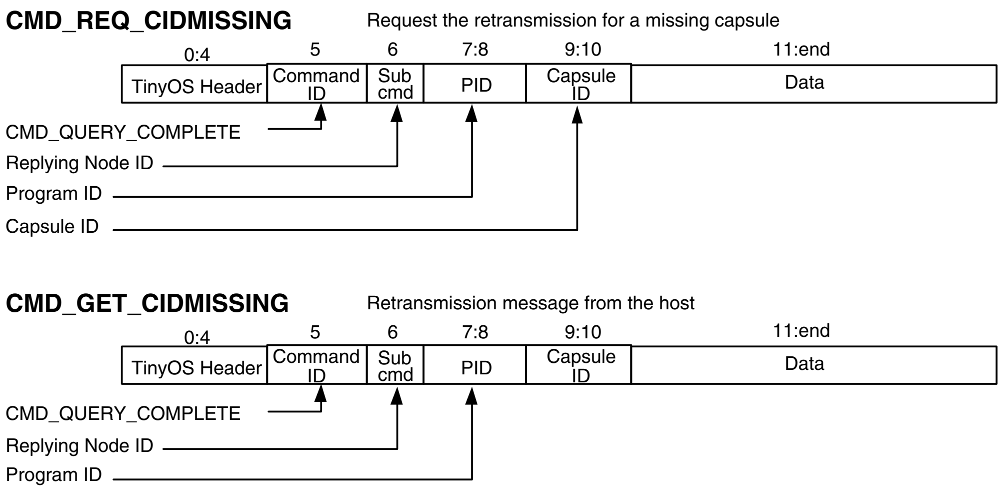


Figure A.1: Network programming message format

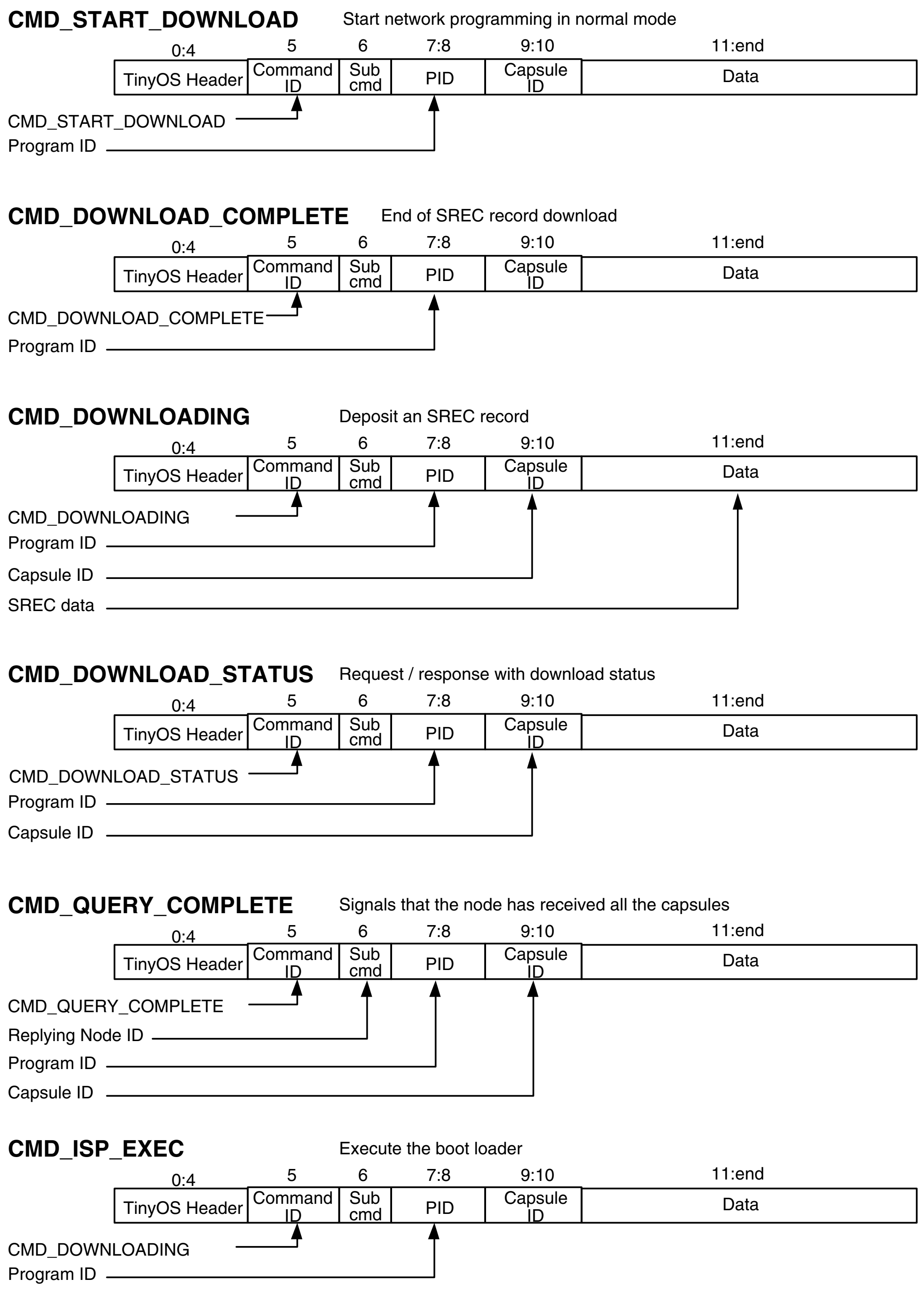


Figure A.2: Network programming message format

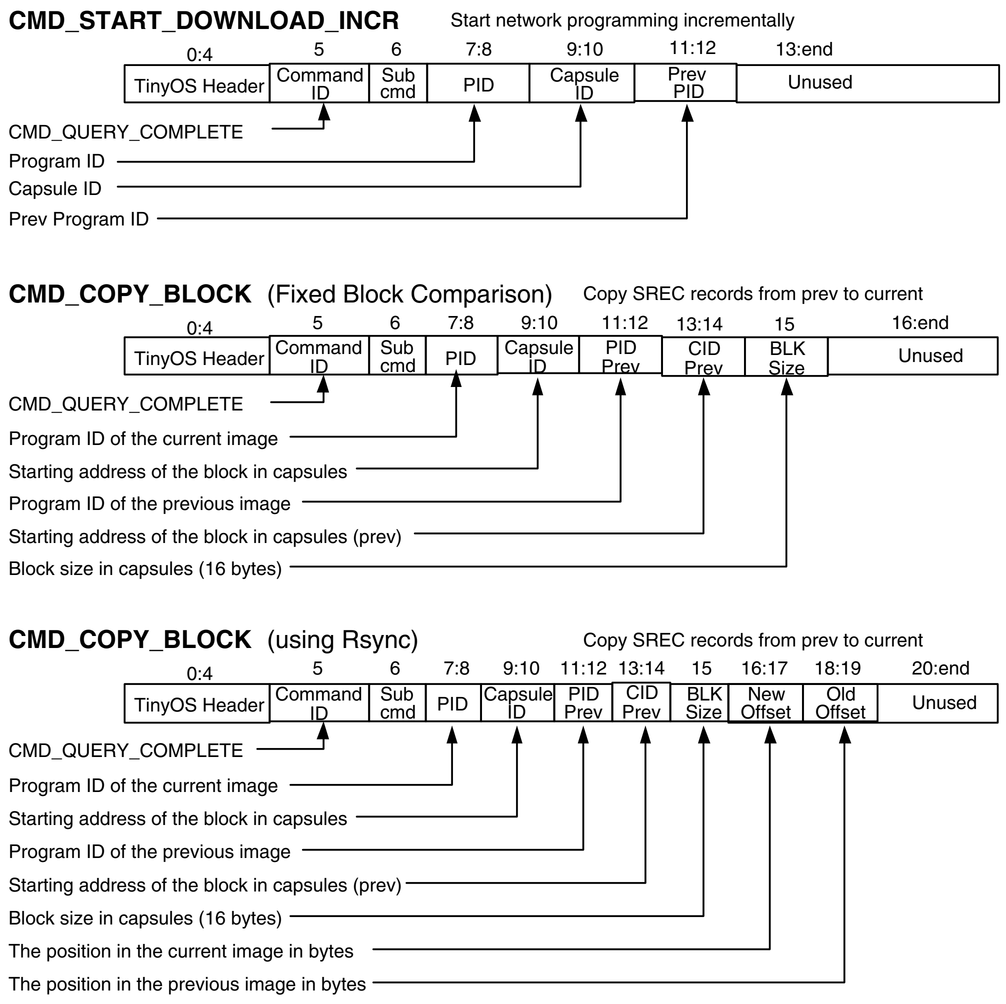

CMD_START_DOWNLOAD_INCR
Start network programming incrementally
0:4
5
6
7:8
9:10
11:12
13:end
TinyOS Header
Command ID
Sub cmd
PID
Capsule ID
Prev PID
Unused
CMD_QUERY_COMPLETE
Program ID
Capsule ID
Prev Program ID
CMD_COPY_BLOCK (Fixed Block Comparison)
Copy SREC records from prev to current
0:4
5
6
7:8
9:10
11:12
13:14
15
16:end
TinyOS Header
Command ID
Sub cmd
PID
Capsule ID
PID Prev
CID Prev
BLK Size
Unused
CMD_QUERY_COMPLETE
Program ID of the current image
Starting address of the block in capsules
Program ID of the previous image
Starting address of the block in capsules (prev)
Block size in capsules (16 bytes)
CMD_COPY_BLOCK (using Rsync)
Copy SREC records from prev to current
0:4
5
6
7:8
9:10
11:12
13:14
15
16:17
18:19
20:end
TinyOS Header
Command ID
Sub cmd
PID
Capsule ID
PID Prev
CID Prev
BLK Size
New Offset
Old Offset
Unused
CMD_QUERY_COMPLETE
Program ID of the current image
Starting address of the block in capsules
Program ID of the previous image
Starting address of the block in capsules (prev)
Block size in capsules (16 bytes)
The position in the current image in bytes
The position in the previous image in bytes

Figure A.3: Incremental network programming message format (Fixed Block Comparison and Rsync)

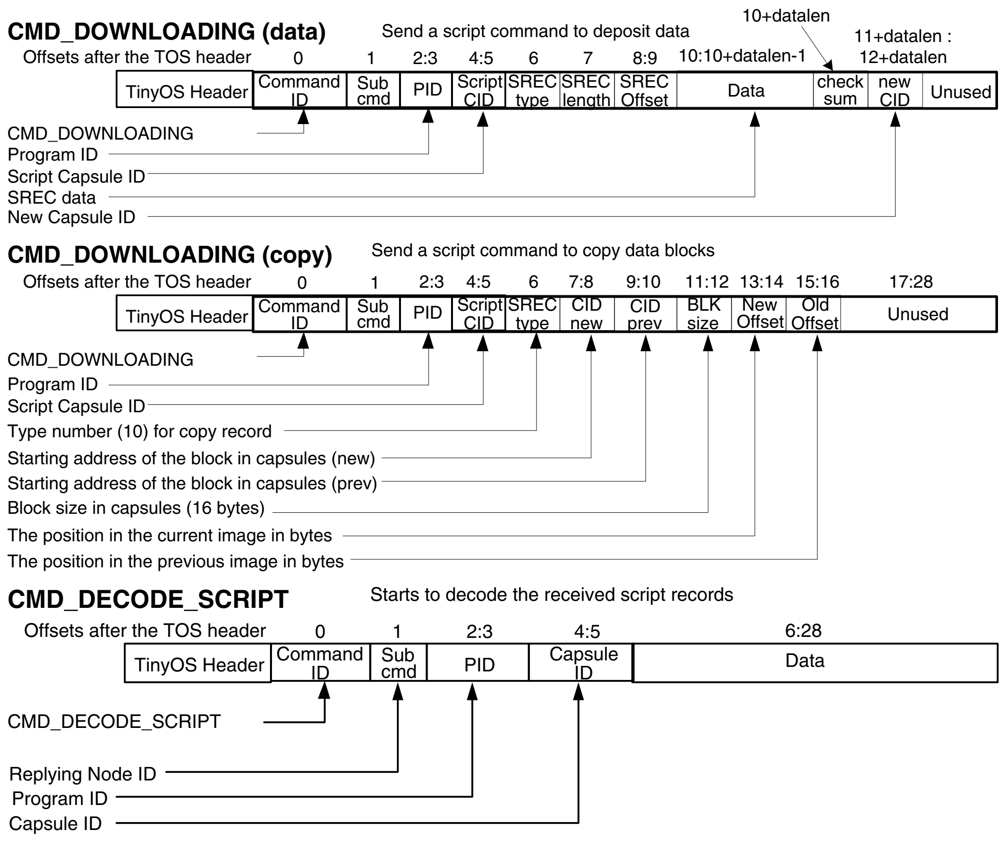


Figure A.4: Incremental network programming message format (Rsync with decode)

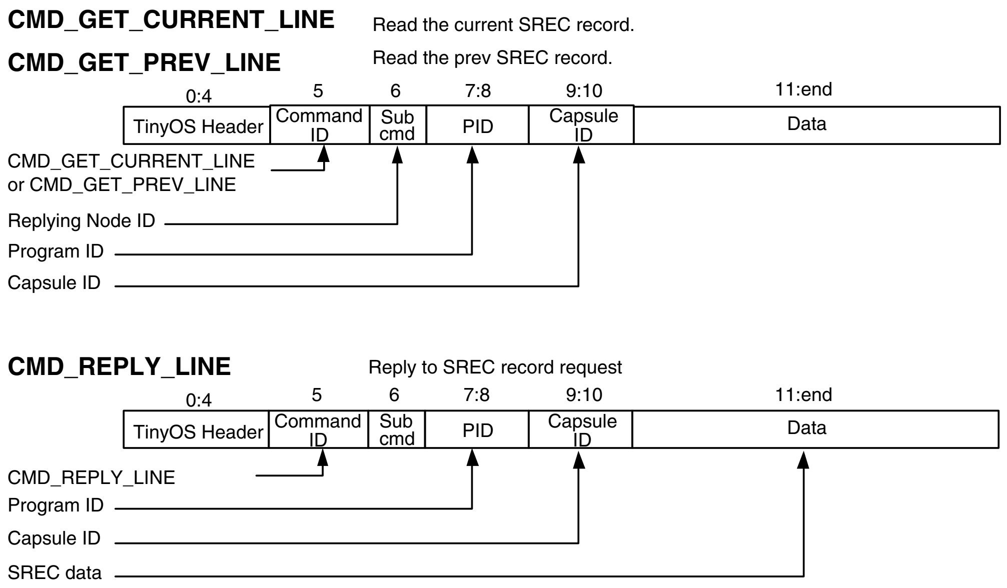


Figure A.5: Network programming message format (debugging messages)

Appendix B

Message Processing

Message Command	Next State	Action
CMD_START_DOWNLOAD	SYS_DL_START	post NPX_STATEMACHINE()
CMD_DOWNLOADING	SYS_DL_SRECWRITE	post NPX_STATEMACHINE()
CMD_DOWNLOAD_COMPLETE	SYS_DL_END	post NPX_STATEMACHINE()
CMD_ISP_EXEC	SYS_ISP_REQ	post NPX_STATEMACHINE()
CMD_GET_CIDMISSING	SYS_REQ_CIDMISSING	post NPX_STATEMACHINE()
CMD_START_DOWNLOAD_INCR	SYS_DL_START_INCR	post NPX_STATEMACHINE()
CMD_COPY_BLOCK	SYS_COPY_BLOCK_PREP	post NPX_STATEMACHINE()
CMD_GET_CURRENT_LINE	SYS_GET_CURRENT_LINE_PREP	post NPX_STATEMACHINE()
CMD_GET_PREV_LINE	SYS_GET_PREV_LINE_PREP	post NPX_STATEMACHINE()

Table B.1: Receiving the incoming message

Start Download

Current State	Next State	Action
SYS_DL_START		fNPXStartDownload() signal Xnp.NPX_DOWNLOAD_REQ()
call from main application Xnp.NPX_DOWNLOAD_ACK()	SYS_DL_START1	post NPX_STATE_MACHINE()
SYS_DL_START1	SYS_DL_START2	call EEPROMWrite.endWrite() post NPX_STATEMACHINE()
SYS_DL_START2	SYS_ACK	post NPX_STATEMACHINE()

Download End

Current State	Next State	Action
SYS_DL_END	SYS_DL_END_SIGNAL	call EEPROMWrite.endWrite() post NPX_STATEMACHINE()
SYS_DL_END_SIGNAL	SYS_ACK	signal Xnp.NPX_DOWNLOAD_DONE() post NPX_STATEMACHINE()

Table B.2: NPX_STATEMACHINE() state transition

Download

Current State	Next State	Action
SYS_DL_SRECWRITE	SYS_EEFLASH_WRITEPREP or SYS_ACK	post NPX_STATEMACHINE()
SYS_EEFLASH_WRITEPREP	SYS_EEFLASH_WRITE	post NPX_STATEMACHINE()
SYS_EEFLASH_WRITE	SYS_EEFLASH_WRITEDONE	post NPX_STATEMACHINE()
SYS_EEFLASH_WRITEDONE	SYS_ACK	call EEPROMWrite.endWrite() post NPX_STATEMACHINE()

Idle

Current State	Next State	Action
SYS_ACK	SYS_IDLE	post NPX_STATEMACHINE()
SYS_IDLE	SYS_IDLE	post NPX_STATEMACHINE()

Retransmission

Current State	Next State	Action
SYS_REQ_CIDMISSING	SYS_GET_CIDMISSING	call EEPROMWrite.endWrite() post NPX_STATEMACHINE()
SYS_GET_CIDMISSING	SYS_GETNEXTCID	post NPX_STATEMACHINE()
SYS_GETNEXTCID	SYS_GETNEXTCID or SYS_GETDONE	post NPX_STATEMACHINE()
SYS_GETDONE	SYS_IDLE	post NPX_STATEMACHINE()

Reprogram

Current State	Next State	Action
SYS_ISP_REQ	SYS_ISP_REQ1	post NPX_STATEMACHINE()
SYS_ISP_REQ1	SYS_ACK	post NPX_ISP()
SYS_DL_START_INCR		fNPXStartDownloadIncr() signal Xnp.NPX_DOWNLOAD_REQ()

Table B.3: NPX_STATEMACHINE() state transition (cont.)

Start Download

Current State	Next State	Action
SYS_DL_START_INCR		fNPXStartDownloadIncr() signal Xnp.NPX_DOWNLOAD_REQ()

Copy Command

Current State	Next State	Action
SYS_COPY_BLOCK_PREP	SYS_COPY_BLOCK_READ	call EEPROMWrite.endWrite() post NPX_STATEMACHINE()
SYS_COPY_BLOCK_READ	SYS_EEFLASH_COPYWRITE	call EEPROMRead.read() fNPXCopyBlk() post NPX_STATEMACHINE()
SYS_EEFLASH_COPYWRITE	SYS_EEFLASH_COPYWRITEDONE	post NPX_wEE_LineWrite() post NPX_STATEMACHINE()
SYS_EEFLASH_COPYWRITEDONE	SYS_COPY_BLOCK_PREP or SYS_ACK	post NPX_STATEMACHINE()

Debugging Commands

Current State	Next State	Action
SYS_GET_PREV_LINE_PREP	SYS_ACK	call EEPROMRead.read() fNPXGetLine() post NPX_STATEMACHINE()
SYS_GET_CURRENT_LINE_PREP	SYS_ACK	call EEPROMRead.read() fNPXGetLine() post NPX_STATEMACHINE()

Table B.4: NPX_STATEMACHINE() state transition (added for incremental network programming)

CMD_DOWNLOADING

Step	Source Lines	Description
`CMD_DOWNLOADING`	29	
`SYS_DL_SRECWRITE`	41	
`SYS_EEFLASH_WRITEPREP`	22	
`SYS_EEFLASH_WRITE`	31	
`SYS_EEFLASH_WRITEDONE`	13	
Total	136	

CMD_COPY_BLOCK (Fixed Block Comparison)

Step	Source Lines	Description
`CMD_COPY_BLOCK`	46	
`SYS_COPY_BLOCK_PREP`	16	Repeated for each SREC line
`SYS_COPY_BLOCK_READ`	40	Repeated for each SREC line
`SYS_EEFLASH_COPYWRITE`	29	Repeated for each SREC line
`SYS_EEFLASH_COPYWRITEDONE`	22	Repeated for each SREC line
Total	153	

CMD_COPY_BLOCK (Rsync)

Step	Source Lines	Description
`CMD_COPY_BLOCK`	46	
`SYS_COPY_BLOCK_PREP`	16	Repeated for each SREC line
`SYS_COPY_BLOCK_READ`	44	Repeated for each SREC line
`SYS_EEFLASH_COPYWRITE`	29	Repeated for each SREC line
`SYS_EEFLASH_COPYWRITEDONE`	22	Repeated for each SREC line
Total	157	

Table B.5: Cost of message handling

Appendix C

Sample Program Source Code

XnpBlink

```
/////////////////////////////////////////////////////////
//  XnpBlink.nc
/////////////////////////////////////////////////////////

configuration XnpBlink {
}
implementation {
  components Main, XnpBlinkM, SingleTimer, LedsC, XnpC;
  Main.StdControl -> SingleTimer.StdControl;
  Main.StdControl -> XnpBlinkM.StdControl;
  XnpBlinkM.Timer -> SingleTimer.Timer;
  XnpBlinkM.Leds -> LedsC;
  XnpBlinkM.Xnp -> XnpC;
}


/////////////////////////////////////////////////////
//  XnpBlinkM.nc
/////////////////////////////////////////////////////

module XnpBlinkM {
  provides {
    interface StdControl;
  }
  uses {
    interface Timer;
    interface Leds;
    interface Xnp;
  }
}
implementation {
  command result_t StdControl.init() {
    call Leds.init();
    call Xnp.NPX_SET_IDS();
    return SUCCESS;
  }
  command result_t StdControl.start() {
    // Start a repeating timer that fires every 1000ms
    return call Timer.start(TIMER_REPEAT, 1000);
  }
  command result_t StdControl.stop() {
    return call Timer.stop();
  }
  event result_t Timer.fired()
  {
    call Leds.redToggle();
    return SUCCESS;
  }
  event result_t Xnp.NPX_DOWNLOAD_REQ(uint16_t wProgramID,
                uint16_t wEEStartP, uint16_t wEENofP){
    // Acknowledge NPX
    call Xnp.NPX_DOWNLOAD_ACK(SUCCESS);
    call StdControl.stop();
    return SUCCESS;
  }
  event result_t Xnp.NPX_DOWNLOAD_DONE(uint16_t wProgramID,
                uint8_t bRet,uint16_t wEENofP){
    if (bRet == TRUE)
       call StdControl.start();
    return SUCCESS;
  }
}
```

XnpCount

```
//////////////////////////////////////////////////////////
//  XnpCount.nc
//////////////////////////////////////////////////////////

configuration XnpCount {
}
implementation {
  components Main, Counter, IntToLeds, IntToRfm, TimerC, XnpCountM,
             XnpC;

  Main.StdControl -> Counter.StdControl;
  Main.StdControl -> IntToLeds.StdControl;
  Main.StdControl -> IntToRfm.StdControl;
  Main.StdControl -> TimerC.StdControl;
  Counter.Timer -> TimerC.Timer[unique("Timer")];
  IntToLeds <- Counter.IntOutput;
  Counter.IntOutput -> IntToRfm;

  Main.StdControl -> XnpCountM.StdControl;
  XnpCountM.Xnp -> XnpC;
  XnpCountM.CntControl -> Counter.StdControl;
}


////////////////////////////////////////////////
//  XnpCountM.nc
////////////////////////////////////////////////

module XnpCountM {
  provides {
    interface StdControl;
  }
  uses {
    interface Xnp;
    interface StdControl as CntControl;
  }
}
implementation {
   uint16_t dest;
   uint8_t  cAck;
  command result_t StdControl.init() {
    call Xnp.NPX_SET_IDS();               //set mote_id and group_id
    return SUCCESS;
  }
  command result_t StdControl.start() {
    return SUCCESS;
  }
  command result_t StdControl.stop() {
    return SUCCESS;
  }
  event result_t Xnp.NPX_DOWNLOAD_REQ(uint16_t wProgramID,
               uint16_t wEEStartP, uint16_t wEENofP){
    //Acknowledge NPX
    call Xnp.NPX_DOWNLOAD_ACK(SUCCESS);
    call CntControl.stop();
    return SUCCESS;
  }
  int dummy(int n)
  {
    int i;
    int sum = 0;
    for (i = 0; i < n; i++) {
      sum += i;
    }
    return sum;
  }
  event result_t Xnp.NPX_DOWNLOAD_DONE(uint16_t wProgramID,
               uint8_t bRet,uint16_t wEENofP){
    // uint32_t ts;
    // call TS.get_timestamp(&ts);
    int temp;
    if (bRet == TRUE)
       call CntControl.start();
    else
       call CntControl.stop();
    temp = dummy(10);
    return SUCCESS;
  }
 }
```