NFC-connected Phone as a User Interface? There's an App For That! – Hands On

Class 5: Putting it All Together

September 27, 2019

Charles J. Lord, PE
President, Consultant, Trainer
Blue Ridge Advanced Design and Automation

Presented by:

DesignNews

Blue Ridge Advanced Design and Automation
Asheville, North Carolina

CEC CONTINUING EDUCATION CENTER

Digi-Key ELECTRONICS

This Week's Agenda

9/23 Introduction to the Project and Development Environment

9/24 An NFC Primer and Introducing the NXP NTAG

9/25 Building an Android Application from Scratch

9/26 Adding NFC Capability and Communications to Our App

9/27 Putting it All Together

This Week's Agenda

9/23 Introduction to the Project and Development Environment

9/24 An NFC Primer and Introducing the NXP NTAG

9/25 Building an Android Application from Scratch

9/26 Adding NFC Capability and Communications to Our App

9/27 Putting it All Together

Quick Review

- If you looked through the two NFC android apps you saw some different implementations of using methods and constructs to read and write simple text strings as NDEF objects.
- We also saw the security methods to check for secure (password protected) and unsecured cards and how to pass a password to unlock a tag/card for read/write.

Question 1: What applications may need encryption?

Beyond Strings

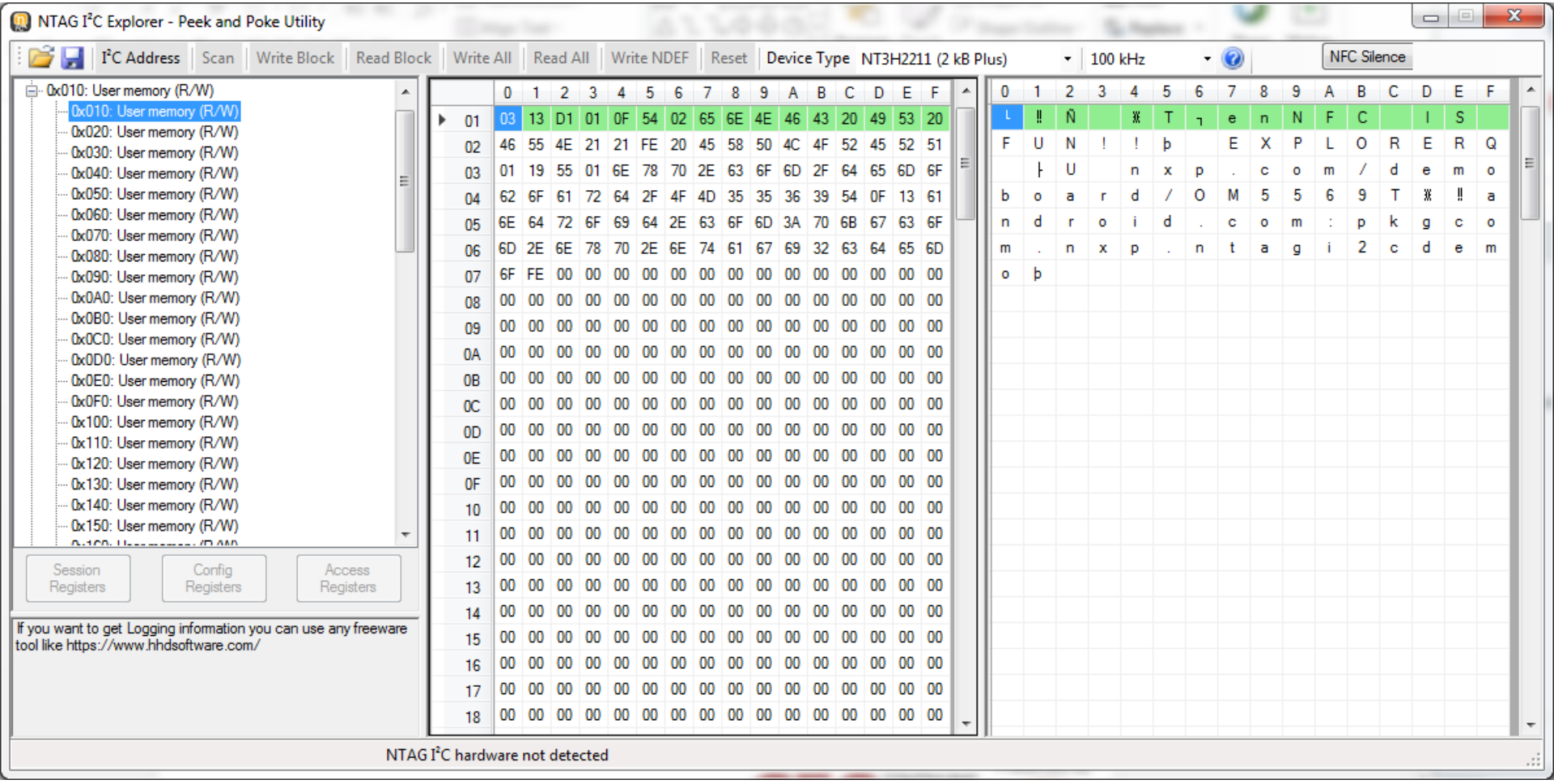
- Of course, with the tags containing 1-2K for the earlier class cards up to 64K in new class 5 cards, we have the capability of much more, up to and including updating the flash of our embedded system via the I^2C port.
- Let's take a moment to actually look at the memory in our NXP NTAG chip.

Peek and Poke

On the “Documents and Software” tab of the NXP OM5569 page that we looked at earlier there is a simple Windows app called Peek and Poke that works with the code in the board’s LPC11U24/401 processor (ARM Cortex M0 from Philips) to control, read, and write to the NTAG chip from the USB port to the I2C port.

https://www.nxp.com/downloads/en/apps/SW3652.zip

Memory Dump (look @ 009-002)



Complete the Path

- We now know a bit about writing an android app and how to use that app to read and write to a NFC tag.
- Now how do we make use of that data to communicate with our embedded application?
- We need to look at the data sheet for our tag chip, the NXP NT322E

https://www.nxp.com/docs/en/data-sheet/NT3H2111_2211.pdf

Near Field Communication CERTIFIED

NT3H2111_2211

NTAG I²C *plus*: NFC Forum T2T with I²C interface, password protection and energy harvesting

Rev. 3.5 — 7 May 2019 Product data sheet
359935 COMPANY PUBLIC

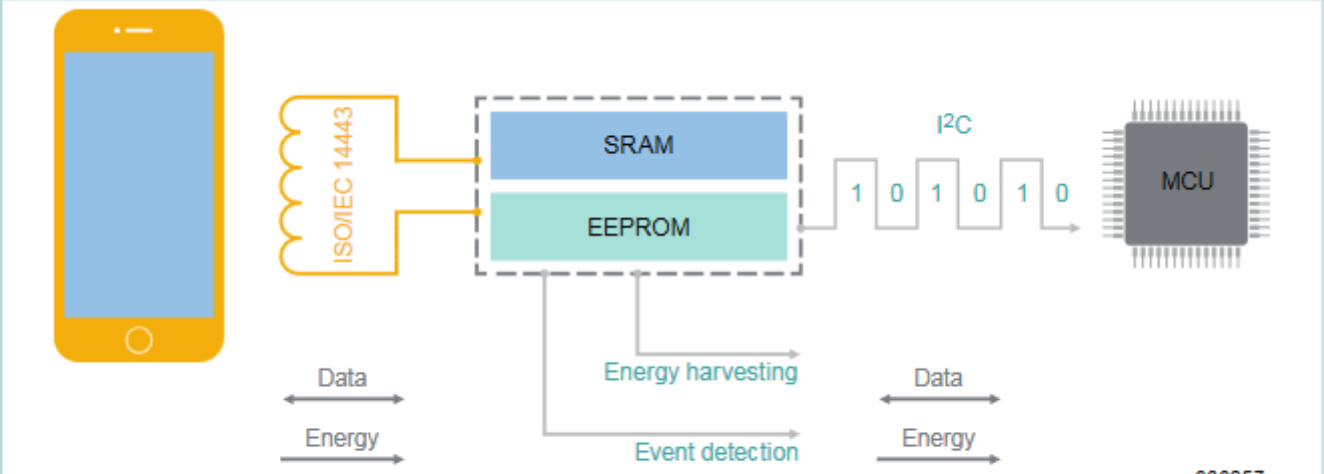


Figure 1. Contactless and contact system

Security

8.7 Password authentication

The memory write or read/write access to a configurable part of the memory can be constrained to a positive password authentication. The 32-bit secret password (PWD) and the 16-bit password acknowledge (PACK) response shall be typically programmed into the configuration pages at the tag personalization stage.

The AUTHLIM parameter specified in Section 8.3.11 can be used to limit the negative authentication attempts.

In the initial state of NTAG I^{2}C *plus*, password protection is disabled by an AUTH0 value of FFh. PWD and PACK are freely writable in this state. Access to the configuration pages and any part of the user memory can be restricted by setting AUTH0 to a page address within the available memory space. This page address is the first one protected.

For a comprehensive description of all protection mechanism refer to Ref. 9.

Remark: The password protection method provided in NTAG I^{2}C *plus* has to be intended as an easy and convenient way to prevent unauthorized memory accesses. If a higher level of protection is required, cryptographic methods can be implemented at application layer to increase overall system security.

10.1 NTAG I²C *plus* command overview

All available commands for NTAG I²C *plus* are shown in Table 16.

Table 16. Command overview

Command[1]	ISO/IEC 14443	NFC FORUM	Command code (hexadecimal)
Request	REQA	SENS_REQ	26h (7 bit)
Wake-up	WUPA	ALL_REQ	52h (7 bit)
Anticollision CL1	Anticollision CL1	SDD_REQ CL1	93h 20h
Select CL1	Select CL1	SEL_REQ CL1	93h 70h
Anticollision CL2	Anticollision CL2	SDD_REQ CL2	95h 20h
Select CL2	Select CL2	SEL_REQ CL2	95h 70h
Halt	HLTA	SLP_REQ	50h 00h
GET_VERSION	-	-	60h
READ	-	READ	30h
FAST_READ	-	-	3Ah
WRITE	-	WRITE	A2h
FAST_WRITE	-	-	A6h
SECTOR_SELECT	-	SECTOR_SELECT	C2h
PWD_AUTH	-	-	1Bh
READ_SIG	-	-	3Ch

Question 2: What other standards apply to NFC?

Every Chip is Unique

8.8 Originality signature

NTAG I²C *plus* features a cryptographically supported originality check. With this feature, it is possible to verify that the tag is using an IC manufactured by NXP Semiconductors. This check can be performed on personalized tags as well.

NTAG I²C *plus* digital signature is based on standard Elliptic Curve Cryptography (ECC), according to the ECDSA algorithm. The use of a standard algorithm and curve ensures easy software integration of the originality check procedure in an application running on an NFC device without specific hardware requirements.

Each NTAG I²C *plus* UID is signed with an NXP private key and the resulting 32-byte signature is stored in a hidden part of the NTAG I²C *plus* memory during IC production.

This signature can be retrieved using the READ_SIG command and can be verified in the NFC device by using the corresponding ECC public key provided by NXP. In case the NXP public key is stored in the NFC device, the complete signature verification procedure can be performed offline.

To verify the signature (for example with the use of the public domain crypto library OpenSSL) the tool domain parameters shall be set to secp128r1, defined within the standards for elliptic curve cryptography SEC (Ref. 10).

Details on how to check the signature value are provided in corresponding application note (Ref. 6). It is foreseen to offer not only offline, as well as online way to verify originality of NTAG I²C *plus*.

Pass-Through is Possible

11 Communication and arbitration between NFC and I²C interface

If both interfaces are powered by their corresponding source, only one interface shall have access to the memory according to the "first-come, first-serve" principle.

In NS_REG, the two status bits I2C_LOCKED and RF_LOCKED reflect the status of the NTAG I²C *plus* memory access and indicate which interface is locking the memory access. At power-on, both bits are 0b, setting the arbitration in idle mode.

In the case arbiter locks to the I²C interface, an NFC device can still read the session registers. If the NFC state machine is in ACTIVE state, only the SECTOR SELECT command is allowed. But any other command requiring EEPROM access like READ or WRITE is handled as an illegal command and replied to with a NAK value.

In the case where the memory access is locked to the NFC interface, the I²C host can still access the session register, by issuing a 'Register READ/WRITE' command. All other read or write commands will be replied to with a NACK to the I²C host.

11.1 Pass-through mode not activated

PTHRU_ON_OFF = 0b (see Table 14) indicates non-pass-through mode.

Pass-Through uses a 64-byte buffer

11.3 Pass-through mode

PTHRU_ON_OFF = 1b (see Table 14) enables and indicates pass-through mode.

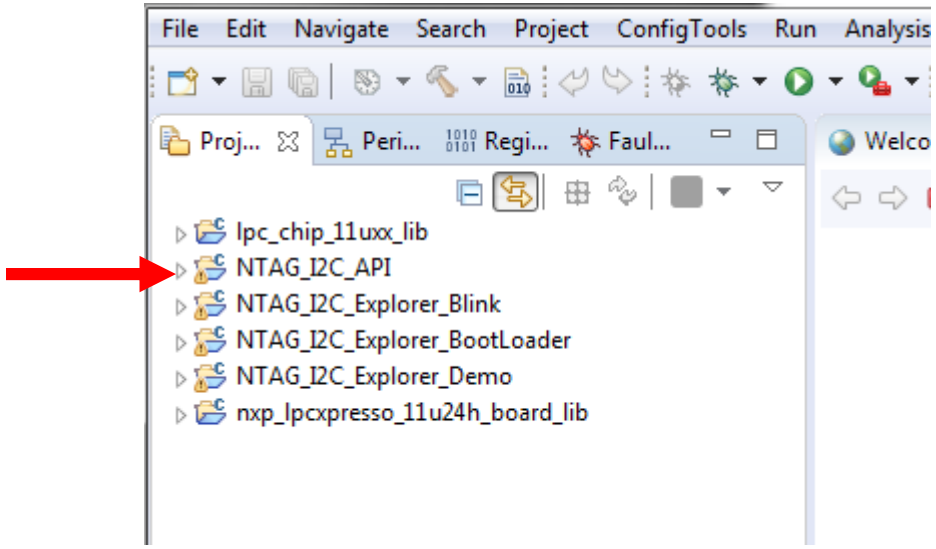
Password protection for pass-through mode may be enabled by enabling password authentication and setting SRAM_PROT bit to 1b.

To handle large amount of data transfer from one interface to the other, NTAG I^2C *plus* offers the pass-through mode where data is transferred via a 64 byte SRAM. This buffer offers fast write access and unlimited write endurance as well as an easy handshake mechanism between the two interfaces.

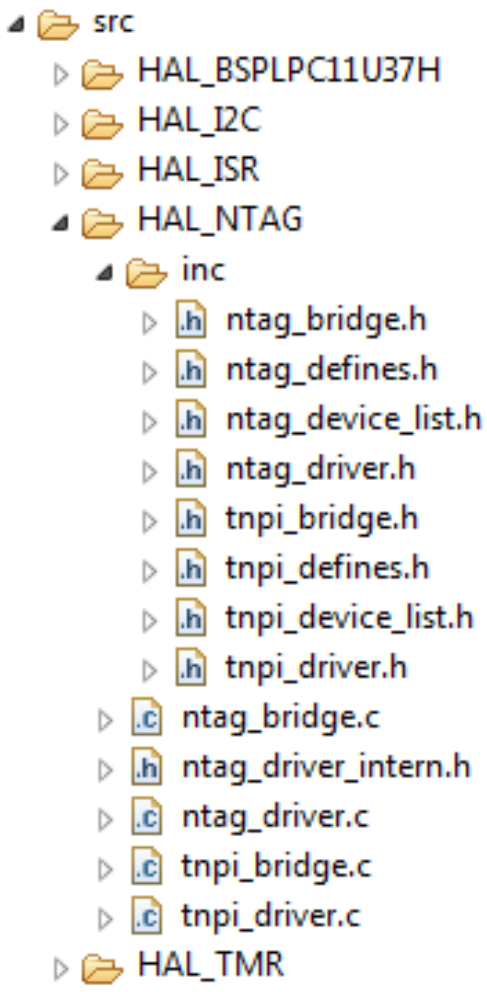
Let's Look at the LPC Source

https://www.nxp.com/downloads/en/snippets-boot-code-headers-monitors/SW3647.zip

- Import this zip file into MCUXpresso IDE
- https://www.nxp.com/docs/en/user-guide/UM10945.pdf



Look at NTAG HAL



Question 3: HAL stands for?

Read

```
98  BOOL NTAG_ReadBytes(NTAG_HANDLE_T ntag, uint16_t address, uint8_t *bytes,
99          uint16_t len) {
100     uint16_t bytes_read = 0;
101
102     if (ntag->status == NTAG_CLOSED)
103         return TRUE;
104
105     ntag->status = NTAG_OK;
106
107     while (bytes_read < len) {
108         uint8_t current_block = (address + bytes_read) / NTAG_I2C_BLOCK_SIZE;
109         uint8_t begin = (address + bytes_read) % NTAG_I2C_BLOCK_SIZE;
110         uint8_t current_len = MIN(len - bytes_read,
111                 NTAG_I2C_BLOCK_SIZE - begin);
112
113         if (current_len < NTAG_I2C_BLOCK_SIZE) {
114             size_t i = 0;
115
116             /* read block into ntag->rx_buffer only */
117             if (NTAG_ReadBlock(ntag, current_block, NULL, 0))
118                 break;
119
120             /* modify rx_buffer */
121             for (i = 0; i < current_len; i++)
122                 bytes[bytes_read + i] = ntag->rx_buffer[RX_START + begin + i];
123         } else {
124             /* full block read */
125             if (NTAG_ReadBlock(ntag, current_block, bytes + bytes_read,
126             NTAG_I2C_BLOCK_SIZE))
127                 break;
128         }
129
130         bytes_read += current_len;
131     }
132     return ntag->status;
133 }
```

```
BOOL NTAG_WriteBytes(NTAG_HANDLE_T ntag, uint16_t address, const uint8_t *bytes,
        uint16_t len) {
    uint16_t bytes_written = 0;

    if (ntag->status == NTAG_CLOSED)
        return TRUE;

    ntag->status = NTAG_OK;

    while (bytes_written < len) {
        uint8_t current_block = (address + bytes_written) / NTAG_I2C_BLOCK_SIZE;
        uint8_t begin = (address + bytes_written) % NTAG_I2C_BLOCK_SIZE;
        uint8_t current_len = MIN(len - bytes_written,
                NTAG_I2C_BLOCK_SIZE - begin);

        if (current_len < NTAG_I2C_BLOCK_SIZE) {
            size_t i = 0;

            /* read block into ntag->rx_buffer only */
            if (NTAG_ReadBlock(ntag, current_block, NULL, 0))
                break;

            /* check if it is the first Block(0x00) and not the I2C Addr */
            /* be careful with writing of first byte in management block */
            /* the byte contains part of the serial number on read but   */
            /* on write the I2C address of the device can be modified    */
            if (0x00 == current_block && NTAG_MEM_ADRR_I2C_ADDRESS < begin)
                ntag->rx_buffer[RX_START + 0] = ntag->address;

            /* modify rx_buffer */
            for (i = 0; i < current_len; i++)
                ntag->rx_buffer[RX_START + begin + i] =
                        bytes[bytes_written + i];

            /* writeback modified buffer */
            if (NTAG_WriteBlock(ntag, current_block, ntag->rx_buffer + RX_START,
            NTAG_I2C_BLOCK_SIZE))
                break;
        } else {
            /* full block write */
            if (NTAG_WriteBlock(ntag, current_block, bytes + bytes_written,
            NTAG_I2C_BLOCK_SIZE))
                break;
        }

        bytes_written += current_len;
    }

    return ntag->status;
}
```

```
227 BOOL NTAG_ReadConfiguration(NTAG_HANDLE_T ntag, uint8_t reg, uint8_t *val) {
228 #ifdef NTAG_2k
229     uint8_t config = NTAG_MEM_BLOCK_CONFIGURATION_2k;
230 #elif NTAG_1k
231     uint8_t config = NTAG_MEM_BLOCK_CONFIGURATION_1k;
232 #endif
233
234     uint8_t I2C_Buf[NTAG_I2C_BLOCK_SIZE];
235     if (NTAG_ReadBlock(ntag, config, I2C_Buf, NTAG_I2C_BLOCK_SIZE))
236         return NTAG_ERR_COMMUNICATION;
237
238     *val = I2C_Buf[reg];
239     return NTAG_ERR_OK;
240 }
241
242 BOOL NTAG_WriteConfiguration(NTAG_HANDLE_T ntag, uint8_t reg, uint8_t mask,
243         uint8_t val) {
244 #ifdef NTAG_2k
245     uint8_t config = NTAG_MEM_BLOCK_CONFIGURATION_2k;
246 #elif NTAG_1k
247     uint8_t config = NTAG_MEM_BLOCK_CONFIGURATION_1k;
248 #endif
249
250     uint8_t I2C_Buf[NTAG_I2C_BLOCK_SIZE];
251     if (NTAG_ReadBlock(ntag, config, I2C_Buf, NTAG_I2C_BLOCK_SIZE))
252         return NTAG_ERR_COMMUNICATION;
253
254     // Clear all other bits of the val
255     val = val & mask;
256
257     // Clear specific bit in the Buffer
258     I2C_Buf[reg] = I2C_Buf[reg] & ~mask;
259
260     // write bits in the Buffer
261     I2C_Buf[reg] = I2C_Buf[reg] | val;
262
263     if (NTAG_WriteBlock(ntag, config, I2C_Buf, NTAG_I2C_BLOCK_SIZE))
264         return NTAG_ERR_COMMUNICATION;
265
266     return NTAG_ERR_OK;
267 }
```

Porting

- For the most part, the code is written to not be device-dependent so that the LPC11 code could be ported fairly easily as long as the chip is supported by MCUXpresso (e.g. Kinetis)
- We then need to determine the functions that we need to carry out for our application to include the proper NFC commands and R/W

Items to Consider

- Use of the chip's unique ID to tie the individual device to the android app
- Security (password) – can be up to 32bits
- How many bytes of data need to be
  - Stored in the flash for possible r/w
  - Transferred at any one time (offload info or set parameters)
- Can data be transferred via strings (or other defined MIME class) – do we need binary?

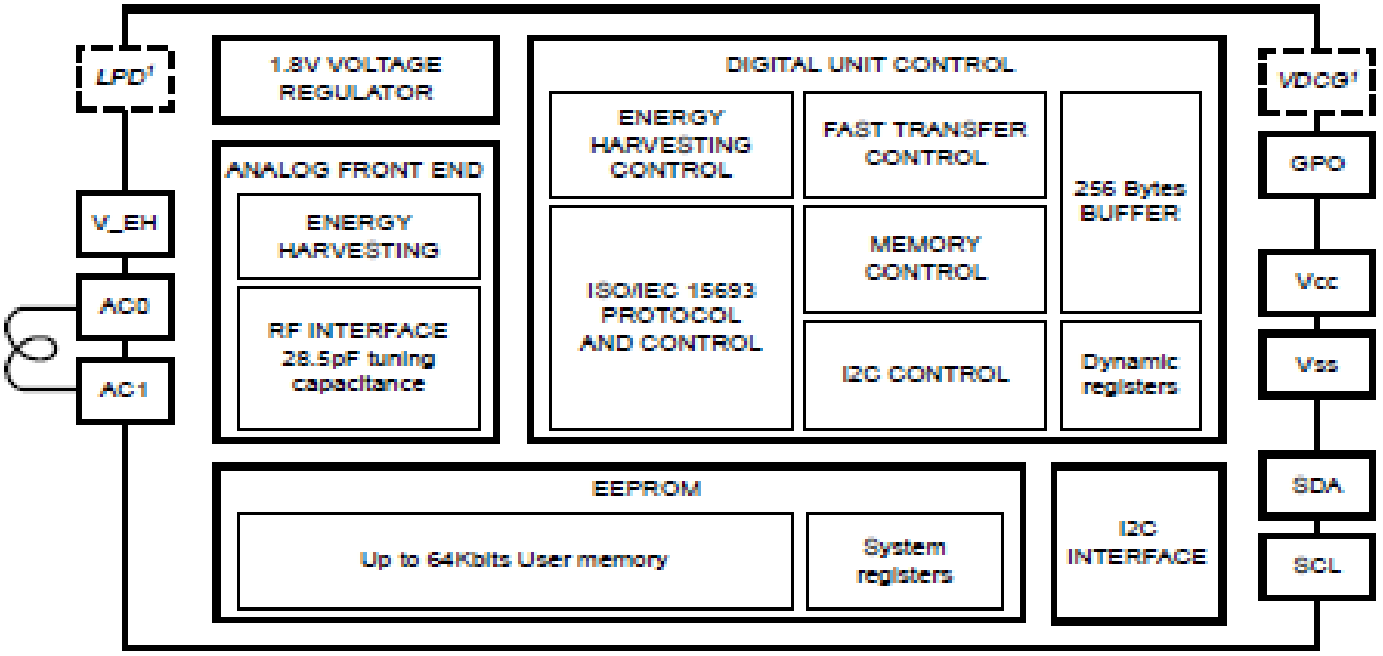
Other items for consideration

- Encryption (requires use of binary)
- Exception handling and fail-safe
  - Out of bounds values
  - Out of bounds R/W (is there rogue data in the flash?)
  - R/W failures
- Does my device work when not in NFC mode, if not can I use energy harvesting?
  - Typical 2V @ 5mA available

Other paths

- Other manufacturers also make NFC tag chips
- ST Micro has a class 5 tag that offers much greater storage and throughput
- Also can consider that STM has the STMCUBE app that can help apply the proper stack for features such as NFC as a click-and-choose
- If you need more data, may consider the ST25DV64K

ST Micro ST25DVxxK



xx = 04, 16, 64 – memory size in kilobytes

Conclusions

- NFC offers a way to add a secure but possibly robust GUI for a device
- This is an excellent way to provide security from tampering while the device is in use
- NFC is not trivial but there are good examples of code to enable the addition to our existing embedded systems
- Hardware need is small (one I^2C port)

Question 4: What new topics would you like to see?

This Week's Agenda

9/23 Introduction to the Project and Development Environment

9/24 An NFC Primer and Introducing the NXP NTAG

9/25 Building an Android Application from Scratch

9/26 Adding NFC Capability and Communications to Our App

9/27 Putting it All Together

Please stick around as I answer your questions!

- Please give me a moment to scroll back through the chat window to find your questions
- I will stay on chat as long as it takes to answer!
- I am available to answer simple questions or to consult (or offer in-house training for your company)
  c.j.lord@ieee.org
  http://www.blueridgetechnc.com
  http://www.linkedin.com/in/charleslord
  Twitter: @charleslord
  https://www.github.com/bradatraining