

Software guide for BitwiseID implementation

Detailed explanation of the Arduino project A_NFC_Scrabble_Board

Scrabble is a registered trademark of HASBRO, Inc

This file is part of NFC Game Board project, presented on www.nfcgameboard.com.

NFC Game Board is free documentaton: you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version.

NFC Game Board is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
See the GNU General Public License for more details.

A copy of the GNU General Public License is published on NFC Game Board.
If not, see <<https://www.gnu.org/licenses/>>.

© 2017-2026 Ben Bulsink

Version 1: June 2017 Initial (Dutch)

Version 2: October 2022 detailing (Dutch)

Version 3: November 2022 renaming and adding flow charts; specify PC-commands (Dutch)

Version 4: April 2024 Google translation to English

Version 4.1: 2026 First publication on www.nfcgameboard.com

TABLE OF CONTENTS

1	Overview	3
2	Scope	3
3	Arduino IDE tabs.....	3
3.1	Coding of identifier in one bit	3
3.2	Set Identifier.....	3
4	Global structure of the Scrabble embedded software	4
5	Data structures.....	6
5.1	Arrays.....	6
5.1.1	RAM located arrays	6
5.1.2	EEPROM tables of the tags in the current set.....	7
5.2	Variables.....	7
5.2.1	EEPROM stored variable	7
6	Tag memory map	7
7	Double storing of tag parameters	8
8	The detection of a set mismatch and solving it	9
8.1	Fresh SetID generation.....	9
9	Detailed explanation of source files and their functions	9
9.1	A_NFC_Scrabble_Board.ino:	9
9.2	Defines.ino.....	9
9.3	Globals.....	9

9.4	Loop.ino	9
9.5	Setup.ino	10
9.6	ScanAndSendChanges.ino	10
9.7	SetManagement.ino	11
9.8	TAGIO.ino	13
9.9	Antennasel.ino	14
9.10	CLRC632.ino	14
9.11	PPP.ino	14
9.12	Messaging.ino	14
9.12.1	Functions:	15
9.12.2	Implemented Single byte commands:	15
9.12.3	Implemented multi-byte commands	15
9.13	Anticollision.ino	16
9.14	DebugMessages.ino	17
9.15	BitwiseXY.ino (_1, _2)	18
9.16	ID_AC.ino	18
10	Scrabble Windows® application	18
10.1	Explanation of the different functions	18
10.1.1	Initialization of a set of tiles	18
10.1.2	Scrabble game reconstruction and points calculation	19
10.1.3	Graphical presentation screen for an ongoing Scrabble game	20

1 Overview

This document describes the Arduino software of the project A_NFC_Scrabble_Board. After reading and studying the source code, it should be possible to understand the structure of the software and make adaptations. In the last chapter, the Windows application ScrabblePresentation is explained briefly.

2 Scope

All the software is found in the Arduino project A_NFC_Scrabble_Board. A few libraries are used: SPI, EEPROM.

The windows application is written in C#, using Microsoft Visual Studio 2022.

Tip: The Arduino project can comfortably be edited, compiled and uploaded using the Visual Micro plugin for Microsoft Visual Studio 2022.

3 Arduino IDE tabs

The sources are divided over a number of tabs, containing separate source files for separate areas of the functionality. A list of the files, the functions they contain and the explanation can be found in chapter 9.

3.1 Coding of identifier in one bit

The BitwiseID approach is invented to be able to identify and make an inventory of a large number of tags within the reading area of an antenna, by only one single ReadMultipleBlocks Icode command.

Each tag identifies itself by having one single bit, unique for that tag, set, in a larger number of bits contained in one or more data blocks, read from the tags.

While all tags respond simultaneously on an non-addressed ReadMultipleblock command, the ICode reading decode procedure causes a logic OR operation of every simultaneous received bit of all simultaneous responding tags. So if each tag only has one unique bit set to logic 1, and the other to logic 0, the ICode reader decodes the received response as a stream of logic 0 and logic 1 bits, where each logic 1 corresponds to a tag that generated this. Actually this is a Time Division Multiple Access (TDMA) method. The approach to code a tag in 1 or more bits is also called Orthogonal Coding.

The amount of bits in which the tags are coded is of course limited. For Scrabble, using 100 tiles, the amount of bits is defined as 128 (4 blocks).

Tags also are identified by the position of the bit they set in the response data. This is actually an assigned identification, assigned within the context of the use of the tag. The context is normally one game set (Set), so a board and 100 tiles. Within this set, the tags receive the identification, being the number of the bit they will set. Identification must be unique, of course, while the tags can not be discriminated, if they set the same bit.

If a tag of another Set, or i.e. a replacement tag, appears within the Set, this must be identified and the tag must be registered and uniquely coded for use in the actual Set.

A Set Management procedure is invented that automatically detects if a tag is not part of the Set, and registers the tag in the Set, and if necessary, re-initialises the Set.

3.2 Set Identifier

The verification of a tile, or better of a tag within this tile, being correctly registered in a Set uses a SetIdentifier or SetID. The SetID is a bit pattern, part of the data read from the tags,

that indicates if all tags that are responding, share the same SetID. The SetID bit pattern is designed as follows:

A SetID bit pattern consists of 20 bits. It is prescribed that each unique SetID bit pattern contains exactly 10 bits being logic 1, and 10 bits, being logic 0.

As the ICode reader decodes the parallel received response of all tags as a logic OR function on the received bits, the read bits of the SetID reveal if all tags carry the same SetID: If two or more different SetID patterns are sent by the tags, the reader will detect more than 10 bits being logic 1. Detecting this, the algorithm has to reregister one or more tags to the current Set.

This re-registering is a complex procedure. It is explained in detail in the chapter that describes the SetManagement functions.

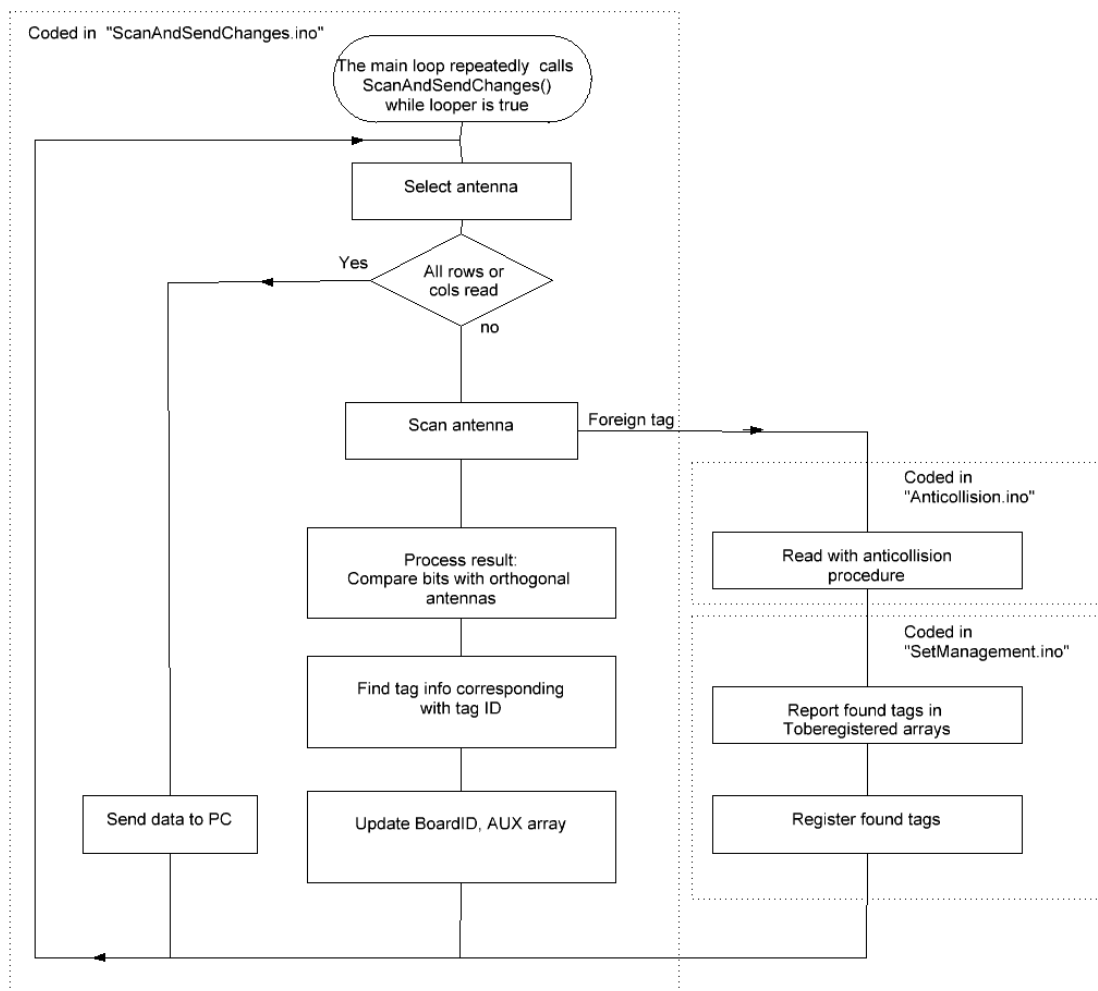
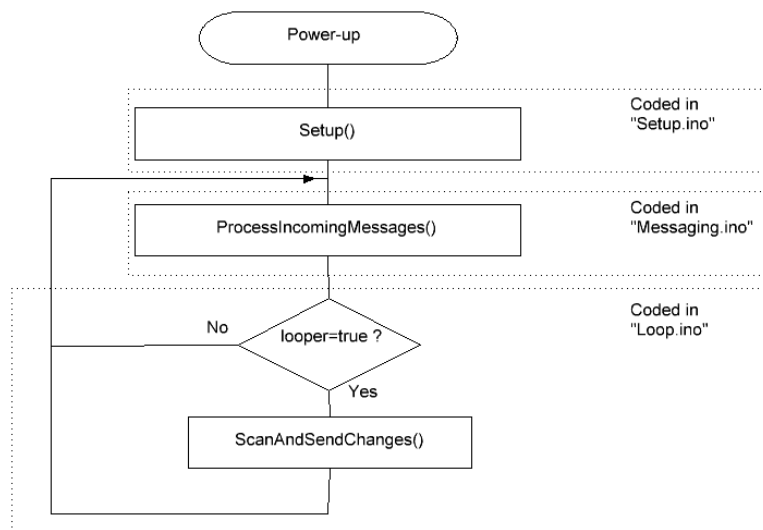
4 Global structure of the A_NFC_Scrabble_Board embedded software

Very important is the administration of the Set Identifier. The SetID value that is in use is saved in EEPROM, and at power-up of the board, this SetID is read and will be used, up to the situation that a new SetID must be defined.

The central function is BitwiseID(). It scans each row and then each column antenna and decodes the tags found by one single ReadMultipleBlock ICode command. The combining of row and column reading information leads to knowledge on which cell tags are found. In case of change compared to the results of a previous scan, the information is sent to the computer.

The racks carry one antenna each, and actually the same reading is done on each rack, which collects the information on which tags are on the racks. This information is also sent to the computer. Racks are scanned more often than the board. (?)

Overview



Verbeteringen:
 Na foreign tag detectie, dezelfde antenne nogmaals scannen, zodat gevonden tags direct verwerkt worden. Nu lijkt dit uitgesteld te worden tot volgende herkenning van die tag (rij of kolom-antenne)

5 Data structures

While the embedded software runs on a relatively small 8-bit microcontroller, in every aspect the economy of the software code must be taken care of. This results in sometimes complex data structures, and given that the BitwiseID procedure is not simple by itself, code can be complex and difficult to read. Where useful, inline comments have been added. To start with the data structures, it is important to constantly realize that the tag UID's are never permanently (during the scanning) stored. Actually, the only moment that the 6-byte UID of a tag is kept in memory is from the moment that a tag of a foreign set is detected (so the SetID of a tag does not match the setID of the current session or game), until this tag has been registered to the actual set, and SetID and BitwiseID values are written to the tag. Also when an Anticollision procedure is requested from the PC, for instance when the set of tiles must be initialized, the UID of tags are kept and sent to the PC.

5.1 Arrays

The more important identification of a tag is its tagID, you could call it a short address. This tagID is assigned to a tag when a tag is detected as foreign. The tagID can range from the value 1 to 127 inclusive, decimal.

The properties of each tag is kept in three 128 byte-arrays in EEPROM (see 5.1.2)

EE_ID_USE[128] contains the registering state of the tags, indexed by the tagID

EE_ID_ROLE[128] contains the letter code of the tag (0=A, 1=B, etc)

EE_ID_AUX contains additional information of each tag: value of the letter(4 bits) and (unused) other information of the tag, i.e. color (4 bits).

Only one RAM located array is indexed with tagID, being TileFromRackNr[128].

Then a few arrays correspond with the rows and columns of the board.

5.1.1 RAM located arrays

byte BoardCR[COLCNT][ROWCNT]

a two-dimensional array containing the ID value of the tags, found on the board. Notice that only one tag at each cell is allowed.

ID or TagIdentifier

The ID of a tag is the bit number of the 128 bit ID field that identifies the tag. Tag with ID 3 has the third bit of the bit field set to logic 1.

byte AuxCR[COLCNT][ROWCNT]

a second array that contains the additional information of the tag on each cell. It contains the points for the letter, code for i.e. the tile color and the number of the rack (= player nr) on which it occurred most recently.

byte BitwiseIDrow[ROWCNT][16] and

byte BitwiseIDcol[COLCNT][16]:

16 bytes (128 bits) for each row and each column which contain the BitwiseID reading result of the antenna.

byte Rack[2][MAXRACKTAGS]

contain the list of tagID's that are found on the two racks.

byte TileFromRackNr[MAXTAGSETSIZE]

This contains the rack number, on which the tag was the last time detected.

5.1.2 EEPROM tables of the tags in the current set

A list of tags that are assigned to the current SetID is maintained in EEPROM.

Where maximum of 128 tags are allowed, the index in the tables represent the tag within the set. I.e. data of a tag with ID 5 is stored in position 5 of the table. The table values are maintained and changed at occurrence of a new tag in a set, or at refreshing of the SetID of the used set of tags.

Table **EE_ID_USE** contains one byte for every tag within the set. The value of the byte in this table defines the information on this tag: ID_UNUSED (= 0) for no tag registered, ID_REGISTERING (=1) for a tag that is in the progress of registering, and ID_INUSE (= 2) for a fully registered tag. ID_INUSE value does not mean that the tag is on the board, only that for this SetID, once a tag is detected and registered with this ID.

Table **EE_ID_ROLE** contains the role of the registered tags, for scrabble the tile letter.

Table **EE_ID_AUX** contains the auxiliary information of the tags: points, color etc. These three tables are written at registering of a tag, and cleared when the SetID is changed, to allow fresh registering of tags.

5.2 Variables

A large number of RAM stored variables are used. See the Globals.ino source file for explanation. In

5.2.1 EEPROM stored variable

One variable, SetID is stored in EEPROM. It contains the 4-byte SetID that is in use.

6 Tag memory map

The R/W memory of a tag is used for several purposes.

First, in normal (registered) use, the area TM_BITWISEID contains 4 x 32 bits of data, which must be understood as an array of 128 bits, where only one bit is set to 1, at the index that corresponds the tagID value.

Then, TM_TAGID1 and TM_TAGID2 both contain the 16-bit SETID of the tag and the human visible information of the tag (Letter, value, tile color).

Additional (non-essential) information is stored or can be stored in TM_RFU, TM_MANU and TM_DATE.

Notice that TM_COL up to TM_ROW2 are reserved for the alternative BitwiseXY detection procedure.

The tag memory content is defined as follows:

Block Number	Name /Function	length (bytes)
0	TM_RFU1 /reserved for future use	4
1	TM_RFU2 /reserved for future use	4
2	TM_MANU /Manufacture code	2
2	TM_DATE /Production date code	2
3	TM_COL /For BitwiseXY: Column code	4
4	TM_ROW /For BitwiseXY: Row code	4
5	TM_COL2 /For BitwiseXY: Copy of column code	4
6	TM_ROW2 /For BitwiseXY: Copy of Row code	4
7	TM_ROLE /16-bits random number + 8 bits Role(letter code), 4 bits letter value and 4 bits for i.e. tag colour	4
8	TM_TAGID1 /SETID (20 bits) + RFU3 (2 bits) + TagWriteIncomplete (1 bit)+ ID (9 bits)	4
9	TM_TAGID2 SETID (20 bits) + RFU3 (2 bits) + TagWriteIncomplete (1 bit) + ID (9 bits)	4
10	TM_BITWISEID /ID: 0x1 << ID. Maximum 128 ID's	16
14	free up to memory end	

Table 1: Tag memory content

The memory usage is a compromise between compactness, for keeping reading speed high, and functionality.

For the A_NFC_Scrabble_Board BitwiseID method, 7 blocks (7-13 inclusive) are read, allowing for 128 tags in a set.

For BitwiseXY, the 5 blocks 3 to 7 inclusive are read. Blocks 0, 1 and 2 are only read for diagnostic purposes.

7 Double storing of tag parameters

During the placement and moving of tags over the sensory board, it can be that the software needs to write data into tags. This happens in BitwiseID, when a tag must be registered to the Set. In BitwiseXY, it also happens when a tag is moved to another cell, or when it appears on the board.

Now, writing to a tag is risky, while it is not sure that it will stay within the energising carrier field during a write operation: It may be that the write operation is not completed. By experiments, it is found that in such cases the result can be that the data is unchanged or that it becomes zero. The Success-bit from the reader is not a guarantee that the data is written correctly. In general, the result of writing to a tag, where there is a risk of moving of the tag out of the field, should always be checked.

Now, there are situations that the SetID block of a tag must be written, to write a new bit pattern. If such writing would fail, and result in all zero bits in this block, it would be impossible to verify if this tag matches the SetID of the board, while it will not show up in the OR operation of the reader.

The solution for this problem is found in having two instances of a value present in the tag, in two different (4-byte) memory blocks. If the one instance would become zero (or is incorrectly written) due to a bad writing, then the other instance still guarantees the reading of the desired value.

Therefor two instances of TM_SetID, TM_COL and TM_ROW are present in the tag memory. For TM_BitwiseID, another mechanism is implemented: Before changing TM_BitwiseID, the TagWriteIncomplete bit in TM_SETID2 is used. This bit is set before TM_BitwiseID is written, and only after verified successful writing of these blocks, TM_TagWriteIncomplete is cleared again.

8 The detection of a set mismatch and solving it

After selecting a new antenna, a ReadMultipleBlocks command is sent to the tags, to return the blocks TM_SetID1 up to all bits used in TM_BitwiseID.

The first check is if the read SetID1 and SetID2 equal the current SetID (in the function `ReadAndRegisterBitwiseID()`), and if TagWriteIncomplete is cleared. If so, the read BitwiseID is reliable for tag localisation.

If a SetID mismatch is found, it must be determined from which tag this is caused. The only way is to do a full Anticollision procedure on reading of the tags on the antenna. Then the tag or tags with the mismatch can be registered to the Set.

This checking and calling of Anticollision is done in the function `Checkmixingup()`.

By using the anticollision procedure, a list of mismatching tags is made, including their unique UID, and when completed, these tags are registered to the Set.

Registering to a Set takes the following steps:

To avoid too much writing of BitwiseID fields, it is checked if the ID of the foreign tag could be used for the Set. This is only possible if the ID is still unused in the current Set. If already used, the first unused ID in the Set is assigned to the foreign tag. Then the block TM_SETID2 is written (after checking that TM_SETID1 is not zero!) with TagWriteIncomplete set. Then TM_BitwiseID is written, and if all OK, TM_SETID1 is written. Finally, TagWriteIncomplete of TM_SETID2 is cleared.

8.1 Fresh SetID generation

To be mentioned: If on an empty board, the first detected tag is not matching the SetID, so foreign, a new SetID is generated at random, and the tag and all future tags are registered to this SetID.

9 Detailed explanation of source files and their functions

9.1 A_NFC_Scrabble_Board.ino:

This first one effectively only includes the libraries and lists the prototypes of all functions. This gives freedom in the sequence of function definition and function call.

9.2 Defines.ino

Contains most #define lines.

9.3 Globals

Contain most of the global variables, available from within any function

9.4 Loop.ino

Contains the standard function `loop()`

Tests on receiving single byte commands, and processes them. A variable loopier is used to allow a command to be repeatedly executed. Temporary structure is used to allow easy adding and changing commands.

These commands should be reformatted in PPP commands and code moved to PPP.ino

9.5 Setup.ino

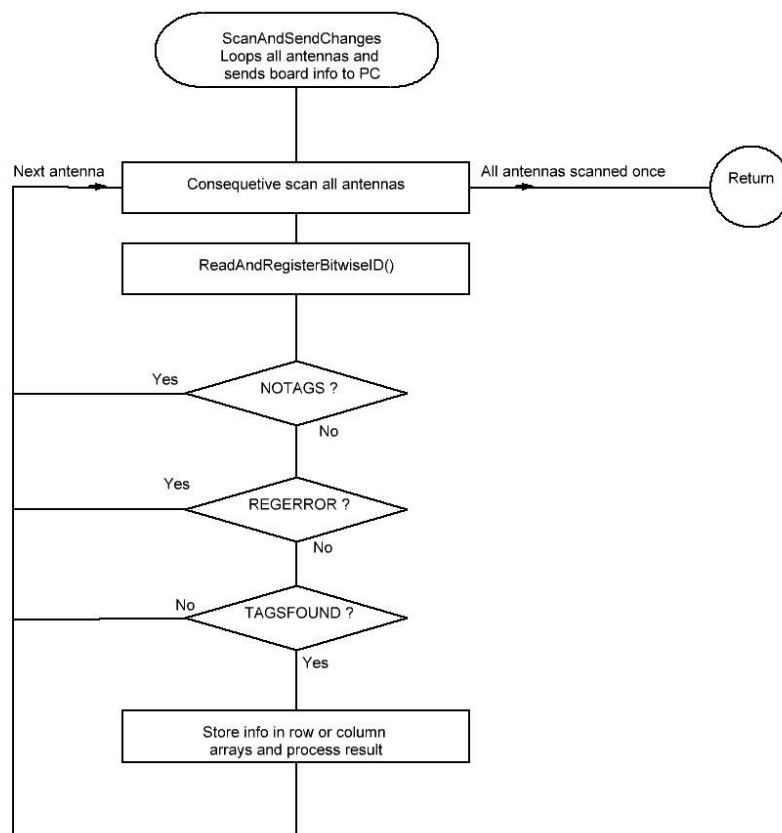
Contains the standard function setup(). Sets pin modes, Initializes SPI, clears contents of arrays, waits for a serial connection, seeds random generator, brings hardware in defined state

9.6 ScanAndSendChanges.ino

The scanning procedure for the Scrabble board 30 antennas + racks.

Functions:

```
byte ScanAndSendChanges();
void updateBoardCR(byte c, byte r, byte newval);
bool IDinBitwiseID(byte byteID, byte *bitwise);
void PPPSendBoardCR();
void PPPSendRack(byte Racknumber);
byte ReadRack(byte Racknumber);
byte ReadAndRegisterBitwiseID();
void SendChangedBoard_ReadRacks();
```



ScanAndSendChanges() scans the board, in two loops, first all rows, then the racks, then all columns. And again the racks.

For each row, column or rack, the function `ReadAndRegisterBitwiseID()` is called. This function causes a reading of a number of blocks from the tags on the antenna, and stores the found BitwiseID tag ID's, which have the size of 128 bits, or 16 bytes, in two arrays `BitwiseIDrow[15]`, `BitwiseIDcol[15]` and `Rack[numberofracks]`. Each array element consists of 16 bytes.

The function `ReadAndRegisterBitwiseID()` checks after each reading of an antenna whether tags are read with foreign SetID values, by calling `CheckMixingUp()`. This complex function also registers foreign tags to the current SetID. See below for explanation of `CheckMixingUp()`.

Now in `BitwiseID()`, when i.e. row antenna row is read, the first step is to check whether the antenna is empty, no tags responding. If so, all the positions on this antenna are made EMPTY in `BoardCR[][row]` and `AuxCR[][row]` is set accordingly. The array element of `BitwiseIDrow[row]` must be cleared, while `ReadAndRegisterBitwiseID()` does not do that (it is not aware of the selected antenna). For changing a board position, the function `void updateBoardCR(c, r, newval)` is used.

Now, as next step, for each square (each column position) in the row, a test is done whether that whole column was empty, the last reading. If so, that square must be empty. Using `updateBoardCR(c, r, newval)` this is then executed.

Now, as the third step, for each found tag in the row (represented by a set bit in `BitwiseIDrow[row]`) the columns reading results `BitwiseIDcol[15]` are checked whether that tag is found in the earlier column reading.

The loop steps through all possible 128 ID's, tests whether the ID is in the row, and if so, tests all columns whether the ID is found there. If found in a column, the tag is updated in the board arrays. If NOT found in the tested column, it is tested whether `BoardCR[col][row]` still contains that ID, and if so, the position is written EMPTY.

Finally, the read information from the antenna is written to `BitwiseIDrow[row]`. (could this be done immediately after reading?)

`CheckMixingUp()` then returns either 0x00 to indicate that no mismatch occurred, or 0x01 when a mismatch was found and tags were registered in the current SetID. `CheckMixingUp()` itself already registers foreign tags.

9.7 SetManagement.ino

This source file contains the functions to detect foreign tags (tags that have not been registered to the current set) and when found, to register foreign tags to the set.

Functions:

```
void SendIDTable(byte amount);
void reporttags_d(byte* start);
unsigned long randomint();
bool CheckSetID(unsigned long SetID);
byte CheckMixingUp();
byte CatchForeignTags(bool uid, byte nrofblocks);
unsigned long randomSetID(unsigned long CurrentSetID);
unsigned long makeTAGID(unsigned long SetID, bool TagWriteIncomplete, unsigned int id);
byte WriteBitwiseID(byte *uid, byte id, byte nrofblocks);
```

```

bool compareSetID(byte *block);
void RegisterNewTags();
byte CatchForReport(bool uid, byte nrofblocks);
void ReportUnexpected(bool uid, byte nrofblocks);
void InitSetID();
void Put_EE_SETID(unsigned long tmpSetID);
unsigned long Get_EE_SETID();
void Put_EE(unsigned int ee_address, byte ee_data);
byte Get_EE(unsigned int ee_address);
void SendEE(int ad, int eelength);
void Addtask(byte task, byte ID, byte *uidx, byte role, byte colorbyte, byte newtagindex);
void NewRandomSetID();
void SetupSetManagement();
byte ReadAndCheckTags();

```

The Set Management is a complicated procedure. The global steps are as follows:

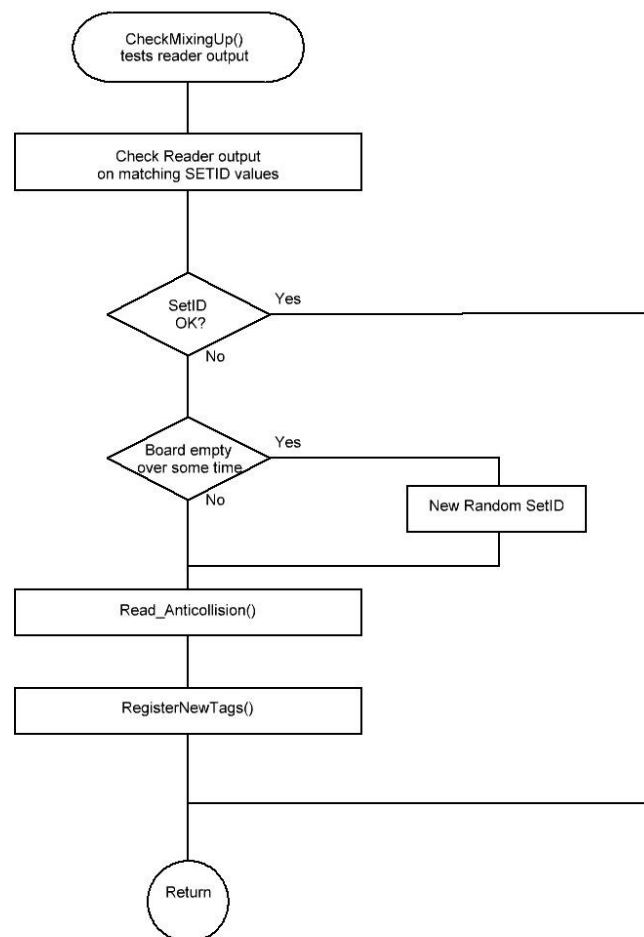
- A. Testing if all tags are part of the set.
When on an antenna, the block of data is read from all present tags simultaneously, the received data of block TM_TAGID1 and TM_TAGID2 are inspected and it is checked whether the SET_ID in these blocks match the SES_ID of the current set. If no foreign tag is found, CheckMixingUp() returns quickly with return value 0.
- B. If the read SET_ID does not fit the SET_ID format, so not 10 bits 1 and 10 bits 0, the tag or tags must be found that cause the mismatch, the foreign tags.
For this, a full inventory of all tags on the antenna must be executed, and each tag must be inspected, whether already part of the set, or foreign. For this, an Anticollision procedure is executed (see Anticollision.ino)
- C. Register foreign tags
At the identification of each tag during the anticollision procedure, it is tested whether the tag is already part of the set, and if not, the tag is registered, by calling RegisterNewTags().
This registering procedure has the following steps:
Check the ID that the foreign tag has. If this ID is unused in the current set, it can be reused.
Now the TagID1, TagID2 and BitwiseID areas of the tag are written. The procedure and sequence is so data integrity is kept (see Double storing of tag parameters), in the situation that the tag gets out of the antenna reach during the writing procedure.

Detailed description

void Read_Anticollision(bool withuid, byte blocknr, byte nrofblocks, byte TagFoundFunction)
This function executes a Anticollision procedure, using slots, mask lengths and mask values. A good explanation of the Anticollision procedure for the 15693 protocol tags is described in the document "Anti-collision on ISO15693 sloa138.pdf" pages 2 and 3.
If on a certain slot number, more than one tags are responding, so causing collision, this slot number and the used mask combined to a new Maskvalue and Masklength, and these values

are added to the corresponding arrays as a new job. The variable lastjobindex is incremented.

Each time when a non-colliding, so correct UID of a tag is found, the function ReportTag is called, which, in case of normal board function, calls CatchForeignTags(). If the data that came from a tag show it is not yet registered, the tag UID is added in the lists ToberegisteredTASK, -ID, -UID, -ROLE and AUX. After the anticollision procedure is completed, the function CheckMixingUp(), that called the anticollision procedure, calls RegisterNewTags(), which processes the list of tags that is found in ToberegisteredUID[].



9.8 TAGIO.ino

The functions that implement the basic tag access functions like Inventory, Block Reading and Block Writing.

Functions:

byte Inv1Slot();

byte ReadMultipleBlocksbyUID(byte *uidx, byte useuid, byte blocknr, byte numberofblocks);

byte WriteSingleBlockbyUID(byte *uidx, bool useuid, byte blocknr, byte *datatowrite);

byte WriteMultipleBlocksbyUID(byte *uidx, bool useuid, byte blocknr, byte nrofblocks, byte *datatowrite); // not supported by many tags

byte WriteSingleBlockbyUIDul(byte *uidx, bool useuid, byte blocknr, unsigned long data);

9.9 Antennasel.ino

Contains the functions to select an antenna on the board.

Functions:

```
void UseRowAntenna(byte row);
void UseColAntenna(byte col);
void SetRowAntenna(byte row);
void SetColAntenna(byte col);
```

9.10 CLRC632.ino

Contains the low level communication with the CLRC632 reader IC, using SPI protocol.

Basic functions are WriteRegister and ReadRegister. The functionality is explained in the CLRC632 specification document. Based on the functions WriteRegister and ReadRegister, a number of more complex operations are built,.

Functions:

```
void InitRegisters_15693();
void TX_on();
void TX_off();
void SendEOF();
void InitRegisters();
void Read_Init();
void Read_Init_shorted();
void FlushFIFO();
void WaitForCLRCReady();
void MFSEL(byte value);
void Write_Init();
void ClearBitMask(byte reg, byte mask);
void SetBitMask(byte reg, byte mask);
void SendToTag(byte data[], byte length);
void WriteRegister(int address, int value);
byte ReadFiFoFast(byte *RBuf);
void RepeatReadFifo(byte address, byte length, byte *ReadBuf);
byte ReadRegister(int address);
```

9.11 PPP.ino

Implements a PPP protocol between Arduino target and computer.

Functions:

```
void SendMessage_PPP (byte header, byte dataformat, byte tobesent[], int length);
void process_ppp_message_in();
void SendDebug(byte debuglevel, char* text);
//void SendDebugF( byte debuglevel, const __FlashStringHelper *text);
```

9.12 Messaging.ino

implements a set of commands that can be called by the computer using a composite PPP format command, both for single byte commands and for commands with parameters.

9.12.1 Functions:

void ProcessMessages();
byte CallbackUIDFound(bool uid, byte nrofblocks);

9.12.2 Implemented Single byte commands:

Name	HEX code	Action	Response to PC:
_0_INVENTORY1SLOT	0x30	Calls Inv1Slot()	FIFO read buffer
1	0x40	looper = true	
2	0x41	ReadRegister(0x03)	Message with status value
3	0x42	looper = false	
4	0x43	Tweak register (ongoing)	
_5_READ_AC	0x44	Read Anticollision	FIFO info of each tag, debug mes nr of tags found
6	0x45	Generates new SESID, then BitwiseID reads RowAntenna(0)	FIFO read buffer and debug messages, then IDtable[0..15]
7	0x46	???	
_8_BITWISE_ID	0x47	Single call to scan board: of ScanAndSendChanges()	
9	0x48	Generates new SESID	
10	0x49	looper = true	
_11_WRITE_EEPROM	0x50	EEPROM write speed test	

Table 2: Single byte PPP commands

9.12.3 Implemented multi-byte commands

The first byte is the command, followed by some byte-parameters (parms).

READBLOCKSUNADDRESSED (0x21) with parms: BLOCKNR, NROFBLOCKS
calls ReadMultipleBlocksbyUID() without UID info

READBLOCKSADDRESSED (0x22) with parms: BLOCKNR, NROFBLOCKS
calls ReadMultipleBlocksbyUID() with UID info

WRITEBLOCKUNADDRESSED (0x23) with parms BLOCKNR, DATA[0-3] , LSB first
calls WriteSingleBlockbyUID() at BLOCKNR with one block DATA

WRITEBLOCKADDRESSED (0x24) with parms BLOCKNR, DATA[0-3](LSB first), UID(LSB first),
calls WriteSingleBlockbyUID() to tag UID at BLOCKNR with one block DATA

SETROWANTENNE (0x25) with parm RowAntenna

SETCOLANTENNE (0x26) with parm ColAntenna

INV_READ_AC_UID (0x27) with parms: BLOCKNR, NROFBLOCKS, ReportYN

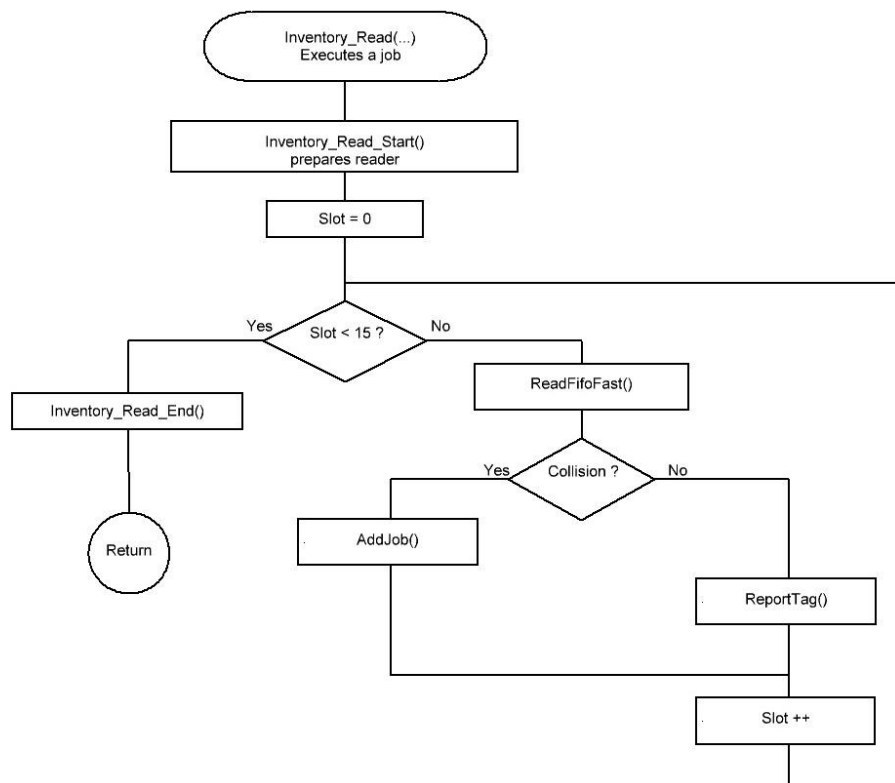
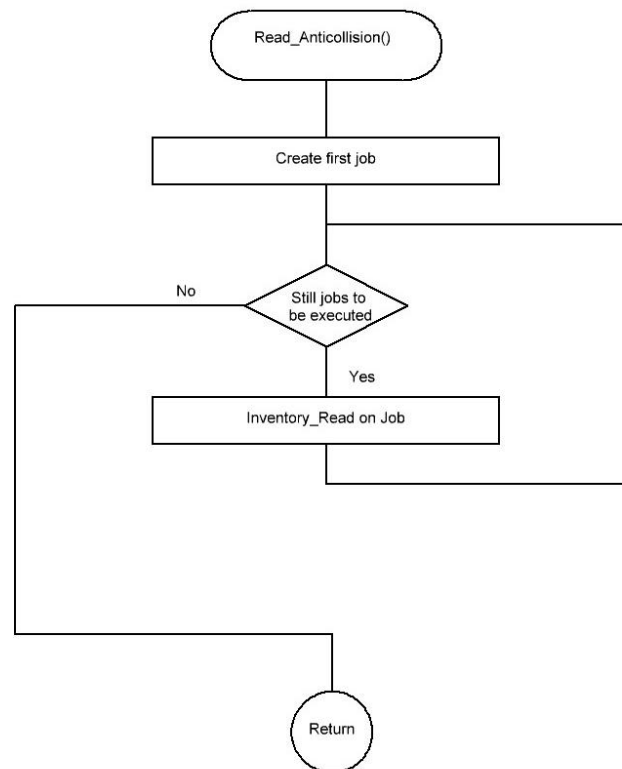
Calls Read_Anticollision() and reports found tags

9.13 Anticollision.ino

The functions for nested anticollision tag detection procedure. Used in SetManagement.

Functions:

```
void Read_Anticollision(bool withuid, byte blocknr, byte nrofblocks, byte Tagfoundf);  
void Inventory_Read(byte currentjob, bool withuid, byte blocknr, byte nrofblocks, byte tagf);  
void Inventory_Read_Start(byte jobpointer, bool withuid, byte blocknr, byte nrofblocks);  
void Inventory_Read_End();  
byte ReportTag(bool withuid, byte nrofblocks, byte desiredfunction);
```



9.14 DebugMessages.ino

Contains a function to send debug messages to the PC.

Features: formats the message in the array buf[] and uses PPP protocol to send to the PC. By using F() for the format string, the format strings are stored in Flash, to save RAM memory.

9.15 BitwiseXY.ino (_1, _2)

The code for using the BitwiseXY scanning procedure. Not yet ported to Arduino. See Visual Studio version. This tab code is not used currently.

9.16 ID_AC.ino

Another non-tested concept of tracking tags.

10 Scrabble Windows® application

The ScrabblePresentation.exe Window application can be used for the following purposes:

- Initialization of a set of tiles
- Scrabble game reconstruction and points calculation
- Graphical presentation screen for an ongoing Scrabble game
- Upload of ongoing Scrabble game to a website for broadcasting
- Debugging functions of the Scrabble board

These functions are still combined in one application, for ease of development, to assure changes to be compatible to all functionality. For future versions, it may be wise to split functions into several applications.

10.1 Explanation of the different functions

10.1.1 Initialization of a set of tiles

At production, tiles are provided with virgin, unprogrammed ICode tags. This eases the manufacturing. Tile letter and letter value can be set to the tag, in a later stage.

To initialize a set of tiles, the first step is to arrange the 100 tiles on a sensor board, in a prescribed manner. I.e. placing them in a 10x10 tags block, sequence A to Z, followed by two blanks, starting at A1 square, up to J10.

```
{ 'A', 'A', 'A', 'A', 'A', 'A', 'A', 'A', 'A', 'B' }  
{ 'B', 'C', 'C', 'D', 'D', 'D', 'D', 'E', 'E', 'E' }  
{ 'E', 'E', 'E', 'E', 'E', 'E', 'E', 'E', 'E', 'F' }  
{ 'F', 'G', 'G', 'G', 'H', 'H', 'I', 'I', 'I', 'I' }  
{ 'I', 'I', 'I', 'I', 'I', 'J', 'K', 'L', 'L', 'L' }  
{ 'L', 'M', 'M', 'N', 'N', 'N', 'N', 'N', 'N', 'O' }  
{ 'O', 'O', 'O', 'O', 'O', 'O', 'O', 'P', 'P', 'Q' }  
{ 'R', 'R', 'R', 'R', 'R', 'R', 'S', 'S', 'S', 'S' }  
{ 'T', 'T', 'T', 'T', 'T', 'T', 'U', 'U', 'U', 'U' }  
{ 'V', 'V', 'W', 'W', 'X', 'Y', 'Y', 'Z', '?', '?' }
```

The correctness of the set is likely, if the right column reads BEFILOQSU?.

Then, the PC application is started. It reads all rows and columns, using anticollision protocol, and so it receives a list of all tag ID's and the row/col coordinates where they were found.

Now, using addressed write operations to the tags, the tag memory content can be written, which includes the letter code, the letter value, eventually something about the tile colour or so, a unique SETID, which is written to all tags of this set, and a BitwiseID value.

Detailing on this process

The following arrays are declared, and used:

ulong[] UIDlist[120] (initialized with value 0x0), will contain the 64 bit UID of found tags

byte[] UIDrow[120] (initialized with value 0xFF), will contain the row number of the corresponding tag in UIDlist

byte[] UIDcol[120] (initialized with value 0xFF), the same, for the col number

Now, for each row antenna of the board, the command INV_READ_AC_UID is send, on which the board will return with messages containing the UID of found tags.

The array UIDlist[] is filled with these received UID's, and on the same index, the row number is written in UIDrow[]

Then, for each column antenna of the board, the command INV_READ_AC_UID is send.

Again, for each found tag, the board will return with a message containing the UID. At each received UID, the array UIDlist is parsed, to find the index of the UID, while that must have been written during row scanning. If found, column number is written at the same index in UIDcol[].

Now, from all 100 tags, the UID's should be listed in UIDlist[], with their row and column written in UIDrow[] and UIDcol[].

Now the writing of all tags is prepared.

A 2-dimensional array Role[rolerow, rolecol] is defined, that contains the role (Scrabble letter) for the tag, according to the prescribed BEFILOQSU? Arranged tiles.

The letter value is found in a pre-set array Value, containing the values from A to Z, and then the blank '?'. A SET_ID is generated (currently 0x1234).

Now, for each of the 100 tags, two tag blocks content is combined from the above mentioned data, being TM_ROLE and TM_TAGID1, and these two blocks are written to the tag, by selecting the corresponding row antenna and invoking WRITEBLOCKADDRESSED commands.

This is the minimum, needed to assure that the tags are recognised by the sensor board, detected as foreign, and completed with fully written TM_TAGID2 and BITWISEID.

Accompanying code lines take care to synchronise the speed and display progress information.

10.1.2 Scrabble game reconstruction and points calculation

The code to process incoming board changes is quite complex. On a certain level, after a stable time, the new positions on the board are evaluated. First, checks are done if the new positions have the structure of being the finalization of a new word placement. This contains i.e. the checks if the new placed tiles are on one row, and the new placed tiles (combined with possible previous placed word tiles) form a range of tiles without a gap.

If all conditions are met, the points of the word are calculated. Then, the new word and the resulting total score is stored and displayed.

Main work of the above evaluation is done by the function `ProcessPosition()`; in the source file `checkpoints.cs`.

First, a check is done on the new placed tiles, if they come from the rack of the player on turn. If OK, then the following steps are executed:

1. Check if all new tiles are on one row or one column.
 - if false: wrong placed or read tiles. exit
 - if true:
2. Check if all new placed tiles join, without gap, eventually with old tiles in between.
Call this extension
 - if false: exit

Now, in function `CheckTouching()`:

3. if no old tiles in between: Check if any new tile has an old tile as direct neighbor
 - if false: exit
4. now check if all words are valid:
 - a. over the extension, if length > 1
 - b. if neighbouring old tiles: check each neighbouring, perpendicular to extension, if extension length > 1
 - if not all valid: Exit

Now in function `CalculateScore()`:

5. Calculate points of all found words
First calculate the new placed tiles row or column. Apply word- and letter multiplication
Then for all other created words:
letter for letter: sum letters points, if new letter is on letter multiplication, multiply letter.
if for the word, one new letter is on word multiplication: multiply sum of letters.
6. Register points
7. Copy position into previous position array.
8. Wait for next turn finished.

Now, the wrap up takes care of setting the turn information, and creates the string for the score display. The function `ChangeDetected()` sends the new game file to internet. Uploading to internet uses the function `client.UploadValues(url, vals)`.

10.1.3 Graphical presentation screen for an ongoing Scrabble game

The creation and updating of the graphical Scrabble board is managed in the source file `board.cs`.

The designer file contains 15 x 15 text box items, one for each board position. The handles are collected in a 2-dimensional array `BD_RC[row,col]`. Then the text box items are filled and coloured, all by calling the function `InitBoard()`.

