BTASM language specification

Jonathan MacMillan (jonathan.macmillan@yale.edu)

Revision 0 - April 4, 2012

Contents

Preface

Introduction

Battle Tag is a laser tag system designed and marketed by UbiSoft. It was released in a limited market (ie, Texas and Canada) near the end of 2010. Since then, the system has not been released in a larger market. It appears to have been abandoned by UbiSoft shortly thereafter, and (as of the creation of this document) none of the equipment can be purchased from UbiSoft's website.

The game itself consists of infra-red "laser" guns, sensor vests, RFID-enabled "bases", and the UbiConnect, a USB peripheral which communicates with the guns using an unknown radio technology. The computer that is connected to the UbiConnect manages all aspects of play (except, of course, those that it offloads to the weapons themselves) and is called the Game Master (henceforth referred to as the GM).

Support for the product seems to be very limited. The new equipment being sold by other vendors in what seems to be a liquidation sale comes with a 90-day manufacturer warranty, which I have no reason to believe will not be honoured. Be sure to thoroughly test the equipment within this 90-day period, as the first sensor vest that I purchased was faulty. There is no easy way to get the equipment in the United States - 1saleaday has been selling vests and guns for around $60 for some time. The expansion kits are nearly impossible to find - I could only source them from Amazon Canada, which does not allow importation of video games to the United States.

Purpose

The purpose of this document is to elucidate the bytecode that is used to control the guns during gameplay. This information has been gleaned from the bytecode and associated source files provided with the BattleTag software. Anyone who wishes to obtain this software must purchase BattleTag equipment - the software is not available legally (to my knowledge) save on the CD-ROM that accompanies the game.

Every time a game is initiated, the GM broadcasts a bytecode to all of the guns that are involved in the game. This bytecode is stored in Lua tables as a comma-separated sequence of bytes. The bytecode appears to be almost completely responsible for the functioning of the device in game - the full workings of each gameplay mode appear to be represented in the bytecode files.

UbiSoft has rather oddly chosen to leave the non-assembled source files for all of the bytecodes in the distributed version of the game. This golden opportunity has allowed reverse-engineering the bytecode to take orders of magnitude less time than it would

have otherwise. The language used seems to be a special language developed in-house at UbiSoft. It is a simple event-driven imperative language, with special language features to control different elements of the weapon. There is a slight bit of work that the assembler needs to do in order to translate between the source code and the bytecode. In fact, it is very straightforward to write bytecode and upload it to the device (by hijacking the game's built-in upload routines).

There appears to be no CRC or consistency check of any sort on the weapons or the GM. In fact, it is perfectly possible to upload broken code to the device. It appears that simply turning the device off and on clears the flashed bytecode and returns the device to a factory state. This, however, has not been thoroughly tested, and it is not recommended that code by written with the intention of interfering with the actual device firmware (though such modification should be possible through editing the GM's Lua files).

The bytecode described here is obviously that used by the weapons. The canonical BTASM language, however, differs from the UbiSoft-style BTASM. The BTASM compiler has a flag (-U) that can parse UbiSoft-style BTASM[1]; canonical BTASM is used by default, and should be used unless there is a particular reason not to do so.

This document does not go into detail concerning the code that the GM must run to keep the game working. This code varies less than one might expect from game type to game type, and is mainly concerned with overall control and scoring. If there is interest, I may cover this information in another document.

Disclaimer

The author makes no guarantee that any of the information presented in this document is accurate. Everything in this document is presented on an "as-is" and "with all faults" basis. The author is not liable for any damage to equipment or persons that is caused by any use, modification, or dissemination of the information presented in this document. You are solely responsible for determining the applicability and compatibility of this information with and to your equipment, and for the protection of yourself and your equipment.

Acknowledgements

Thanks to BL Martech, who alerted me to the fact that the Battle Tag distribution included Lua source files. Thanks, also, to the developers at UbiSoft, who left a veritable roadmap for us to follow. All thanks, of course, are given to God, who granted me a mind and a will to better understand this part of His creation. May this work glorify Him alone.

[1] This flag is included mainly for testing against UbiSoft-style files

1 The beginning

1.1 Environment

BTASM is whitespace-agnostic. Control structures are defined explicitly, though indentation is recommended in order to facilitate comprehension of control structures. BTASM is case-sensitive; identifiers beginning with an uppercase letter are all reserved. Variables must begin with a lowercase letter and can contain alphanumeric charaters and the underscore character.

1.1.1 UbiSoft-style

UbiSoft-style BTASM source files are very simple in nature. They consist of:

- A variable declaration block
- Optional function declarations
- A required initial state
- Optional additional states

These parts of a source file must appear in this order in an UbiSoft-style BTASM file.

The variable declaration block consists of a newline-separated list of variable expressions which define a variable by name and determine its relationship to the GM. All variables are **32-bit integers** (CHECK THIS).

By default, all variables are declared as global variables in the scope of the gun program. The scope of the variable as it relates to the GM can be modified with the following keywords:

- CONFIG (???)
- SEND (the variable is sent to the GM in regular updates)???
- RECEIVE (the variable is sent to the gun from the GM in regular updates)???

Variables of the different types can be declared in any order. No variable can be declared with more than one attribute type: thus `VAR foo SEND, CONFIG` is an invalid declaration. All of these variables exist in a global namespace. There are no limited-scope variables. Variables cannot be declared outside of the variable declaration block.

```
VAR a    SEND
VAR b
VAR c    RECEIVE
VAR d    CONFIG
VAR e    SEND
```

Figure 1.1: A valid variable block in an UbiSoft-style BTASM program

The optional function declarations follow the variable declaration block. A function definition consists of a special keyword (FUN) and a name. There is no separation function declaration - the function definition serves as the declaration. Therefore, function declarations *must/should* be placed at the beginning of the source file/before they are first used in the program. **The variable and function name-spaces do not conflict (THIS NEEDS TO BE TESTED!!) If no functions are needed, than none need be declared. Note that these functions are decidedly non-functional, since they depend on global values, and neither accept arguments or return values. They would be better referred to as procedures – following the syntax of the UbiSoft-style language, we will refer to them as functions.

```
FUN foo
    HUD_ICON_ON BULLET
    HUD_ICON_ON LIFE
END_FUNCTION
```

Figure 1.2: A valid function definition in an UbiSoft-style BTASM program

Event states are the main drivers of the gun's action. The control flow of a BTASM program can be conceptualized as a state machine. The gun begins the game in an initial state (denoted by `FIRST_STATE`), in which it responds to environmental stimuli. At points, the gun can switch states, and respond differently to the same stimuli. This paradigm will be familiar to anyone who has programmed embedded devices.

The possible events that a particular state can respond to are (at least): (nb - check the bytecodes to see if there are more)

- `ANIM_FINISHED` - finished HUD animation
- `BUTTON_1_JUST_PRESSED` - pressed button below HUD
- `BUTTON_2_JUST_PRESSED` - pressed trigger
- `BUTTON_3_JUST_PRESSED` - pressed RFID button
- `DATA_CHANGE` - GM has changed a variable declared with the `RECEIVE` attribute
- `ENTER_STATE` - Code run upon entering a state

- `HIT` - hit by another gun
- `TIMER` - ??probably responds to end of TIMER that was set earlier
- `TICK` - ??do something on each tick of the clock (what clock?)

Each of these events, when called, runs a block of code which can do anything that can be done in a function. It can `GOTO` an alternate state; call a function; change variables; or simply do nothing. Events cannot be declared more than once per state.

The state namespace does not conflict with the function namespace or the variable namespace. It is possible to have a variable named foo, a function named foo, and a state named foo (this is, however, not recommenced, and the compiler will print a warning if it detects double-naming).

```
STATE foo
FIRST_STATE

    EVENT BUTTON_1_JUST_PRESSED
        HUD_ICON_ON BULLET
    END_EVENT

    EVENT BUTTON_3_JUST_PRESSED
        HUD_ICON_OFF BULLET
    END_EVENT

END_STATE
```

Figure 1.3: A valid state definition in an UbiSoft-style BTASM program

```
VAR a SEND
VAR b

FUN foo
    HUD_ICON_ON BULLET
    HUD_ICON_OFF LIFE
END_FUNCTION

FUN bar
    HUD_ICON_ON LIFE
    HUD_ICON_OFF BULLET
END_FUNCTION

STATE baz
FIRST_STATE

    EVENT BUTTON_1_JUST_PRESSED
        foo     --call function foo
    END_EVENT

    EVENT BUTTON_3_JUST_PRESSED
        bar     --call function bar
    END_EVENT

END_STATE
```

Figure 1.4: A valid UbiSoft-style BTASM program

2 Language definition

This chapter contains a definition of each of the operations in the BTASM bytecode, along with an explanation of their operation. This chapter explains the bare mechanics of the bytecode, not the syntax and semantics of the associated UbiSoft-style BTASM language. The author recommends that people use the proposed canonical syntax that maps more closely between bytecode and language.

2.1 Instruction set

2.1.1 Declarations

0xcc - VAR [0,1,2,3]

Declares a global variable

- 0 = no attributes
- 1 = `SEND`
- 2 = `RECEIVE`
- 3 = `CONFIG`

0xd0 - FUN [FUNIDX] [INT]

Declares a function

- A1 = function identifier (non-repeated ascending integers)
- A2 = size of function in bytes (maximum value of 256 - not counting declaration)

NB - this is an overloaded instruction that is also used for function calls with a different type signature

0xd2 - STATE [LONGINT] [STATEIDX]

Declares a new (initial) state

- A1 = size of the state in bytes (not counting declaration)
- A2 = state identifier (non-repeated ascending integers)

NB - There can be only one initial state per program.

0xc7 - STATE [LONGINT] [STATEIDX]

Declares a new (non-initial) state

- A1 = size of the state in bytes (not counting declaration)
- A2 = state identifier (non-repeated ascending integers)

NB - Though there are different bytecodes for the initial state and a regular state, the state identifiers must be unique for each different state.

[EVENTCODE] - EVENT [EVENTCODE] [INT]

Declares a new event in a state. Event types:

- `0x00 = BUTTON_1_JUST_PRESSED`
- `0x01 = BUTTON_2_JUST_PRESSED`
- `0x02 = BUTTON_3_JUST_PRESSED`
- `0x09 = TIMER`
- `0x0a = TICK`
- `0x0b = HIT`
- `0x0d = ENTER_STATE`
- `0x0e = ANIM_FINISHED`
- `0x0f = DATA_CHANGE`

The second argument is the length of the event code in bytes (not counting declaration)

2.1.2 Control flow

0xc3 - GOTO [STATEIDX]

Switch to a different state. That state's ENTER_STATE event will be executed.?????

- A1 = index in state table to move to

0xc4 - IF [VARIDX] 0x01 [LONGINT] [INT] [IF branch len]...[ELSE branch len]...

0xc4 - IF [VARIDX] 0x00 [VARIDX] [INT]...[IF branch len]...[ELSE branch len]...

Compare two values and branch depending on the result of the comparison.

- A1 = index in variable table of left-side comparison variable
- A2 = 0x01 for immediate comparison; 0x00 for variable comparison
- A3 = immediate value to compare or index to variable to compare with (right side)
- A4 = comparison function (0 = SUP; 2 = COMP; 3 = DIFF)
- A5 = length of the IF branch of the conditional
- A6 = length of the ELSE branch of the conditional (can be 0x00)

NB - the two length arguments are placed just before the code that their associated branches contain. The language has no JMP capabilities, so the IF and ELSE branches must follow the IF instruction. The ... in the instruction definition represent source code. There is no need to specify the ending point of a branch - the length arguments handle the length.

0xd0 - CALL [FUNIDX]

Call a function.

- A1 = index in function table of function to call