**Software Implementation and Integration**

27 March 2015

2IO70

Version 1

The purpose of this document is to show the data representation and coding standard we chose that is used to write the Assembly Language.

Group 16

Rolf Verschuuren

Wigger Boelens

Stefan van den Berg

Dat Phung

Maarten Keet

Tudor Petrescu

**Table of contents**

[Java to PHP 3](#_Toc415260069)

[Validation of PHP to Java 3](#_Toc415260070)

[Validation of Assembly to PHP compiler 3](#_Toc415260071)

[Appendix 2: Explanation of the compiler functions 5](#_Toc415260077)

# Java to PHP

The Java to PHP conversion is usually natural, the two languages sharing most syntax but there are some differences we must note down. We are not required to create a class in PHP. The initialization will differ in PHP from Java, but they share the same core in the end. Also while we have some of the variables initialized globally in Java, in PHP they will be local. Having no class will make the class initialization irrelevant in PHP and that’s why its missing. The later functions in the Java code right after the function TimerManage are included in the PHP code using “include “functions.php”;”. In TimerManage, % operation is replaced by the mod() function. Due to our PHP compiler limitations we are required to use variables as arguments when calling functions.

# Validation of PHP to Java

Because of the natural similarity and ease of conversion, the PHP codes correctness can be correlated to its java counterpart, the correctness of the java code was validated in the Validation part of the Software Design.

# Validation of Assembly to PHP compiler

The compiler works in phases. We will go through these phases 1 by 1 to explain how the compiler does its job: compiling PHP-like code to assembly. Throughout the phases the compiler keeps track of the line number of the PHP code it is currently compiling and uses that, when an error occurs, to give information where the error is. The compiler is written in PHP5.6 and uses a command line interface.

## Preprocessing

In the first phase, the input code will be made ready for the next steps. A few things happen in this phase: First the file is read into the memory. The next step is that all comments, newlines and extra spaces are stripped from the file. The file is then split into single lines using the “;” symbol that denotes the end of a line. While doing this the compiler writes the data to two arrays: the data array for everything between “//\*\*DATA\*\*” and “//\*\*CODE\*\*” and the code array for everything after “//\*\*CODE\*\*”. Everything before //\*\*DATA\*\* is ignored. The data array gets compiled immediately.

The preprocessor further removes some special statements that are needed to make valid php such as “global” and changes some shortcuts in their full version. For example $abc++ will be changed into $abc+=1. This ensures that the compiler only needs to be able to handle $abc+=1.

## Splitting

In the second phase the code is split up by function. Every function gets his own array with all the lines that are in that function. The code not inside of a function goes into a separate array.

## Compiling

The third phase is the most important one. It starts by compiling the code that is at the start and not inside a function. While compiling it keeps track of what functions are called and adds these, if they are not already compiled, to the toCompile queue. This helps in making sure there is no dead code, as a function that is never called, will not be compiled. The compiler adds the function “main”, which is the default start point of the code, to the queue and starts processing it.

After compiling the main function it will continue in the next function in the toCompile queue and keep doing this till the toCompile queue is empty.

The compiling itself is not a lot more than a lot of regex and switch statements that look at the input and make an output from that. At the first notion of a variable a register is assigned to it. The code then uses this register in place of the variable. Some more difficult statements, like the function display which displays something, will BRS to premade assembly code that handles that. The compiler keeps track of which segments of the premade assembly code are used.

When the compiler meets an if statement, it saves the code inside it to a new function named “condtionali” where i is the amount of conditionals that have already been seen. It then places this function in the toCompile queue. It also saves the location of the end of the if statement, so it will later know where to return when the if function has ended.

## Combining

After there are no functions left in the toCompile queue, the combining phase starts. In this phase all the functions and the code outside the functions are combined into a single array. This phase also adds the used premade functions at the top and inserts the return statements at the correct position.

## Formatting

The last phase is the last interesting. It goes through the, now compiled code, and formats it. It uses either the length of the longest function name or the number 25 depending on which is larger to insert spaces in front of every line of code in a way everything lines up nicely.

The last step the compiler takes is writing the compiled code to a file and using the assembler provided to create the hex code.

# Appendix 1: Explanation of the compiler

The compiler works in phases. We will go through these phases 1 by 1 to explain how the compiler does its job: compiling PHP-like code to assembly. Throughout the phases the compiler keeps track of the line number of the PHP code it is currently compiling and uses that, when an error occurs, to give information where the error is. The compiler is written in PHP5.6 and uses a command line interface.

**Preprocessing**

In the first phase, the input code will be made ready for the next steps. A few things happen in this phase: First the file is read into the memory. The next step is that all comments, newlines and extra spaces are stripped from the file. The file is then split into single lines using the “;” symbol that denotes the end of a line. The code is divided in three segments. The first segment starts at //\*\*COMPILER, everything before this statement is ignored.

The preprocessor further removes some special statements that are needed to make valid php such as “global” and changes some shortcuts in their full version. For example $abc++ will be changed into $abc+=1. This ensures that the compiler only needs to be able to handle $abc+=1.

**Splitting**

In the second phase the code is split up by function. Every function gets his own array with all the lines that are in that function. The code not inside of a function goes into a separate array.

**Compiling**

The third phase is the most important one. It starts by compiling the code that is at the start and not inside a function. While compiling it keeps track of what functions are called and adds these, if they are not already compiled, to the toCompile queue. This helps in making sure there is no dead code, as a function that is never called, will not be compiled. The compiler adds the function “main”, which is the default start point of the code, to the queue and starts processing it.

After compiling the main function it will continue in the next function in the toCompile queue and keep doing this till the toCompile queue is empty.

The compiling itself is not a lot more than a lot of regex and switch statements that look at the input and make a output from that. At the first notion of a variable a register is assigned to it. The code then uses this register in place of the variable. Some more difficult statements, like the function display which displays something, will BRS to premade assembly code that handles that. The compiler keeps track of which segments of the premade assembly code are used.

When the compiler meets an if statement, it saves the code inside it to a new function named “condtionali” where i is the amount of conditionals that have already been seen. It then places this function in the toCompile queue. It also saves the location of the end of the if statement, so it will later know where to return when the if function has ended.

For every line it compiles, it takes the corresponding line of PHP and inserts it as a comment in the assembly. This is to help in debugging.

**Combining**

After there are no functions left in the toCompile queue, the combining phase starts. In this phase all the functions and the code outside the functions are combined into a single array. This phase also adds the used premade functions at the top and inserts the return statements at the correct position.

**Formatting**

The last phase is the least interesting. It goes through the, now compiled code, and formats it. It uses either the length of the longest function name or the number 25 depending on which is larger to insert spaces in front of every line of code in a way everything lines up nicely. It also makes sure the comments line up nicely.

The last step the compiler takes is writing the compiled code to a file and using the assembler provided to create the hex code.

# Appendix 2: Explanation of the compiler functions

storeRam($location, $value)

Store a value in the ram.

$location The location (a variable) to store the value in the ram

$value The value to store, needs to be a variable

return void

getRam($location)

Get a value from the ram.

$location The location (a variable) where the value is stored

return The value that is stored at the location

display($what, $onWhat, $location = '000001')

Display something on either the display or the leds.

Possible values for $onwhat:

* leds: the leds at the top
* leds2: the leds to the right
* display: the display

$what What to display, must be a variable

$onWhat On what to display

$location Where to show the value when using the display, defaults to the right position

return void

pow($number,$power)

Get the power of a number

$number The number to power

$power The power value

return Int; The result

mod($what, $variable)

Take the modulo of a number

$what Modulo what

$variable Variable to modulo over

return void

getInput($writeTo, $type)

Get button or analog input. When you just want the input of 1 button, use getButtonPressed instead.

$writeTo Variable to write the input to

$type Type of input, possible values are: buttons, analog

return void

getButtonPressed($button)

Check if a button is pressed. Puts the result into R5.

$button Which button to check (input a variable)

return Int; Whether or not the button is pressed.

installCountdown($functionName)

Install the countdown.

$functionName The name of the function where the timer should go to

return void

startCountdown()

Start the countdown.

Retrun void

pushStack($variable)

Push a variable to the stack

$variable The variable to push to the stack

return void

pullStack($variable)

Pull a variable from the stack.

$variable The variable where the pulled variable is put into

return void

setCountdown($countdown)

Set the timer interrupt to a value. It will first reset the timer to 0.

$countdown How long the countdown should wait, in timer ticks

return void

getData($location, $offset)

Get data. Use offset 0 when it is just a single value.

$location The location where the variable is stored

$offset The offset of the location

return The value of the data segment

storeData($variable, $location, $offset)

Store data. Use offset 0 when it is just a single value.

$variable The variable to store

$location The name of the location where the variable is stored

$offset The offset of the location

return void

sleep($howLong)

Pause the program.

$howLong How long to sleep in clockticks

return void

initVar($variable,$places)

Initialize a variable that is used in that data segment.

$variable The name of the variable

$places How long the array is

return void

branch($branchTO)

Branch to a function.

$branchTO where to branch to

return void

moveFunction($branchTO)

Move a function in the assembly code.

$branchTO Where to branch to

return void