CS271 Style Guide

This guide is intended to describe an assortment of styling practices when coding. These rules may be requirements for formatting your assignments, but they also serve to improve your program's readability and maintainability. Good organization keeps the bugs away. Special thanks to Joshua Cox for his work on this document.

Oregon State University
College of Engineering

Alignment & Indentation Rules

Indenting and aligning your code and comments makes it easier to read, helping you to make changes and spot mistakes.

Alignment

No variable alignment

```
buffer BYTE BUFFER_SIZE dup(?) ; temporary array for input
inputMsg BYTE "Enter a number between 1-50", 0ah, 0dh, 0
resultMsg BYTE "The generated string is: ", 0
error1 BYTE "Cannot create file", 0ah, 0dh, 0
fileName BYTE "RandomString.txt", 0
capital BYTE 0 ; boolean to check for capitals
redStr BYTE "red", 0
blueStr BYTE "blue", 0
greenStr BYTE "green", 0
```

Variables and Comments Aligned

```
buffer      SWORD BUFFER_SIZE dup(?)             ; temporary array for input
inputMsg    BYTE  "Enter a number between 1-50", 0ah, 0dh, 0
resultMsg   BYTE  "The generated string is: ", 0
error1      BYTE  "Cannot create file",
fileName    BYTE  "TestString.txt", 0
capital     DWORD 0                              ; boolean to check for capitals
redStr      BYTE  "red", 0
blueStr     BYTE  "blue", 0
greenStr    BYTE  "green", 0
```

No Instruction Alignment

```
main PROC
  MOV EDX,offset inputMsg
  CALL WriteString

  ; get input
  call ReadInt
  MOV ECX,10
  MOV EBX,0

  _Add:
    ADD EAX,EBX
      INC EBX
      LOOP  _Add

  CALL CrLf
  CALL DisplayResult
  exit
main ENDP
```

Instructions Aligned Together

```
main PROC
  MOV    EDX, OFFSET inputMsg
  CALL   WriteString

  ; get input
  CALL   ReadInt
  MOV    ECX, 10
  MOV    EBX, 0

_Add:
  ADD    EAX, EBX
  INC    EBX
  LOOP   _Add

  CALL   CrLf
  CALL   DisplayResult
  exit
 main ENDP
```

Indentation

All instructions and comments should be offset from the left by a TAB OR two/four SPACES, while labels are aligned to the left. We won't force tabs over spaces or the reverse.

No indentation

```
; prepare array for printing
MOV  ECX, LENGTHOF intArray
MOV  EDI, OFFSET intArray
MOV  EAX, 0

_Read:
; print next element

MOV  EAX, [EDI]              ; set EAX to next element of intArray
CALL WriteInt
CALL Crlf
ADD  EDI, TYPE intArray      ; increment to next element
LOOP _Read
```

Offset Indenting

```
  ; prepare array for printing
  MOV  ECX, LENGTHOF intArray
  MOV  EDI, OFFSET intArray
  MOV  EAX, 0

_Read:
  ; print next element

  MOV   EAX, [edi]           ; set EAX to next element of intArray
  CALL  WriteInt
  CALL  Crlf
  ADD   EDI, TYPE intArray   ; increment to next element
  LOOP _Read
```

Syntax Rules

Identifiers

Identifiers should be descriptive, explaining what they're used for without requiring a user to search their uses to understand them. camelCase is preferred.

Undescriptive vs.

```
.data
  x          DWORD 5
  y          DWORD 1
  n          DWORD 0

.code
  MOV    ecx, x
_L1:
  ADD    n, y
  INC    y
  LOOP   L1
```

Descriptive

```
.data
  count            DWORD 5
  addSource        DWORD 1
  sum              DWORD 0

.code
  MOV    ecx, count
_SumLoop:
  ADD    sum, addSource
  INC    addSource
  LOOP   _SumLoop
```

Constants

Constants should be declared above the *.data* segment and written in all capitals.

Mixed, Lowercase vs.

```
.data
count = 12
limit = 100
array DWORD count dup(?)
msg   BYTE "Enter a number.", 0
```

Separated, Uppercase

```
COUNT = 12
LIMIT = 100

.data
  array  DWORD COUNT dup(?)
  msg    BYTE "Enter a number.", 0
```

Labels

Labels should be aligned to the left, offset from lines of instructions. They should be written any way except uppercase, and preferably with a leading underscore (_) so they are not mistaken for some form of constants.

Unaligned, Uppercase vs.

```
USERINPUT:
CALL   ReadInt
MOV    [ESI], eax
ADD    ESI, TYPE bye
LOOP   USERINPUT
QUIT:
```

Offset, not Uppercase

```
_UserInput:
  CALL   ReadInt
  MOV    [ESI], eax
  ADD    ESI, TYPE bye
  LOOP   UserInput
_Quit:
```

Oregon State University College of Engineering

Back to Top

Comment Rules

Comments are a vital means for describing the intent of your code, not only to others but to your future self. Assembly especially requires careful commenting due to its particularly unfriendly syntax. Different comment types help to convey different information, but all of them attempt to do the same thing: explain what the program is doing and why. This guide covers four different comment types: *In-line*, *Block*, *Section*, *Procedure comments* (headers), and *Macro Comments* (headers).

Program Header

The program header must describe (in your own words) the general program functionality. It also must include the key identifying information present in the example below (Author Name, Last Modified, OSU Email Address, Course Number/Section, Project Number, Due Date).

```
TITLE Guessing Game One Shot     (GuessingGameOneShot.asm)

; Author: Stephen Redfield
; Last Modified:  09/22/2020
; OSU email address: ONID_ID@oregonstate.edu
; Course number/section:   CS 271 Section ???
; Project Number: NULL              Due Date: 09/22/2020
; Description: Program asks user to pick a value between 1 and 10.
;     If user gets the correct number (defined as constant), print congratulations.
;     If not, notify the user they have lost the game.

INCLUDE Irvine32.inc
; ...
```

In-line Comments

Occasionally a single instruction might be particularly complicated, or important to the behavior of the program and require extra attention. Because they're used to highlight certain lines of the program, overusing in-line comments may cause important comments to become lost in a sea of unhelpful comments.

Too many unhelpful in-line comments

```
  ; display numbers
  MOV   ECX, lengthof intArray       ; set LOOP counter to length of intArray
  MOV   EDI, offset intArray         ; set edi to Addr. of intArray
  MOV   EAX, 0                       ; clear eax
_Read:
  MOV   EAX, [EDI]                   ; set eax to next element of intArray
  CALL  WriteInt                     ; print element of intArray
  CALL  Crlf                         ; print clear line
  ADD   EDI, type intArray           ; increment to next element
  LOOP  _Read                        ; continue LOOP
```

Fewer, but helpful in-line comments

```
  ; display numbers
  MOV   ECX, lengthof intArray
  MOV   EDI, offset intArray
  MOV   EAX, 0
_Read:
  MOV   EAX, [EDI]                   ; set EAX to next element of intArray
  CALL  WriteInt
  CALL  Crlf
  ADD   EDI, type intArray           ; increment to next element
  LOOP  _Read
```

Block Comments

In almost every situation, a single task will require multiple lines of code: validating a number, reading and saving input, or fetching and printing some value. Following the Separation of Concerns, code that works to complete a single discrete task should be grouped together with a clear purpose. It's up to you to decide whether a block of code should be separated down into smaller descriptive blocks.

Fewer, large blocks vs.

```
; print intro and greet user
MOV   EDX, OFFSET introduction
CALL  WriteString
MOV   EDX, OFFSET instructions
CALL  WriteString
MOV   EDX, OFFSET nameMsg
CALL  WriteString
MOV   EDX, OFFSET userName
MOV   ECX, NAME_SIZE
CALL  ReadString
MOV   EDX, OFFSET greet
CALL  WriteString
MOV   EDX, OFFSET userName
CALL  WriteString
CALL  Crlf
```

More, smaller blocks

```
; print intro and instructions
MOV   EDX, OFFSET introduction
CALL  WriteString
MOV   EDX, OFFSET instructions
CALL  WriteString

; get user's name
MOV   EDX, OFFSET nameMsg
CALL  WriteString
MOV   EDX, OFFSET userName
MOV   ECX, NAME_SIZE
CALL  ReadString

; greet user
MOV   EDX, OFFSET greet
CALL  WriteString
MOV   EDX, OFFSET userName
CALL  WriteString
CALL  Crlf
```

Section Comments

Blocks of code can be further grouped into sections, where multiple blocks work together to achieve a single greater task. Breaking code into sections becomes less common as procedures are used, allowing code to be more logically separated into discrete procedures. Because section comments describe many blocks of code, they should try to summarize the purpose of that section of blocks.

Not Descriptive, no Summary of Purpose

```
; Get User Data
  MOV    EDX, offset message
  CALL   WriteString
  ; ...
```

Descriptive summary

```
; --------------------------
; Asks user for positive integers and stores them
;     in an array, then checks they are in the valid range.
;     If they are invalid, prints an error.

; --------------------------
  MOV    EDX, OFFSET message
  CALL   WriteString
  ; ...
```

Procedure Comments/Headers

As programs become more complex, sections are usually replaced by procedures to improve flexibility and reusability. Unlike other comments, procedure comments, or procedure headers, act like very brief instruction manuals, describing how a piece of code is used, and how it works. They have multiple components to help explain how the procedure works: *description, preconditions, postconditions, receives*, and *returns*.

Procedure Header Template

```
; ---------------------------------------------------------------------------------
; Name: procedureName
;
; The description of the procedure should be like a section comment, summarizing
;     the overall goal of the blocks of code within the procedure.
;
; Preconditions: Preconditions are conditions that need to be true for the
;     procedure to work, like the type of the input provided or the state a
;     certain register need to be in.
;
; Postconditions: Postconditions are any changes the procedure makes that are not
;     part of the returns. If any registers are changed and not restored, they
;     should be described here.
;
; Receives: Receives is like the input of a procedure; it describes everything
;     the procedure is given to work. Parameters, registers, and global variables
;     the procedure takes as inputs should be described here.
;
; Returns: Returns is the output of the procedure. Because assembly procedures don't
;     return data like high-level languages, returns should describe all the data
;     the procedure intended to change. Parameters and global variables that the
;     procedure altered should be described here. Registers should only be mentioned
;     if you are trying to pass data back in them.
; ---------------------------------------------------------------------------------
procedurename PROC
; ...
```

Procedure Header Example 1

If the procedure uses parameters, describe each parameter in *receives* and any conditions they have in *preconditions*.

```
; ---------------------------------------------------------------------------------
; Name: findSmallest
;
; Finds the smallest integer in an array and returns it in the eax register.
;
; Preconditions: the array contains only positive values.
;
; Postconditions: none.
;
; Receives:
;     [ebp+16]     = type of array element
;     [ebp+12]     = length of array
;     [ebp+8]      = address of array
;     arrayMsg, arrayError are global variables
;
; returns: eax     = smallest integer
; ---------------------------------------------------------------------------------
findSmallest PROC USES ESI ECX EAX
; ...
```

Procedure Header Example 2

Describe any named parameters in *receives*. It isn't necessary to thoroughly describe local parameters.

```
; ---------------------------------------------------------------------------------
; Name: arrayPrint
;
; Prints an integer array in rows of 10 elements.
;
; Preconditions: the array is type DWORD.
;
; Postconditions: changes registers esi, ecx, edx
;
; Receives:
;    ptrArray = address of array
;    sizeArray = length of array
;
; returns: none
; ---------------------------------------------------------------------------------
arrayPrint PROC,
  ptrArray: PTR DWORD,
  sizeArray: DWORD,
  LOCAL counter: DWORD
; ...
```

Procedure Header Example

Simple procedures with any components that are *none* can be omitted.

```
; ---------------------------------------------------------------------------------
; Name: goodbye
;
; Displays a goodbye message.
;
; Receives:
;     [ebp+8]     = reference to message
; ---------------------------------------------------------------------------------
goodbye PROC
; ...
```

Macro Comments/Headers

Similar to procedure headers, Macro headers should briefly describe their use and requirements. Macro names should begin with a lowercase “m” to distinguish them from procedures and variables.

Macro header example

```
; ---------------------------------------------------------------------------------
; Name: mGenerateString
;
; Generates a random string of lowercase letters.
;
; Preconditions: do not use eax, ecx, esi as arguments
;
; Receives:
; arrayAddr = array address
; arrayType = array type
; arraySize = array length
;
; returns: arrayAddr = generated string address
; ---------------------------------------------------------------------------------
mGenerateString MACRO arrayAddr:REQ, arrayType:REQ, arraySize:REQ
; ...
```