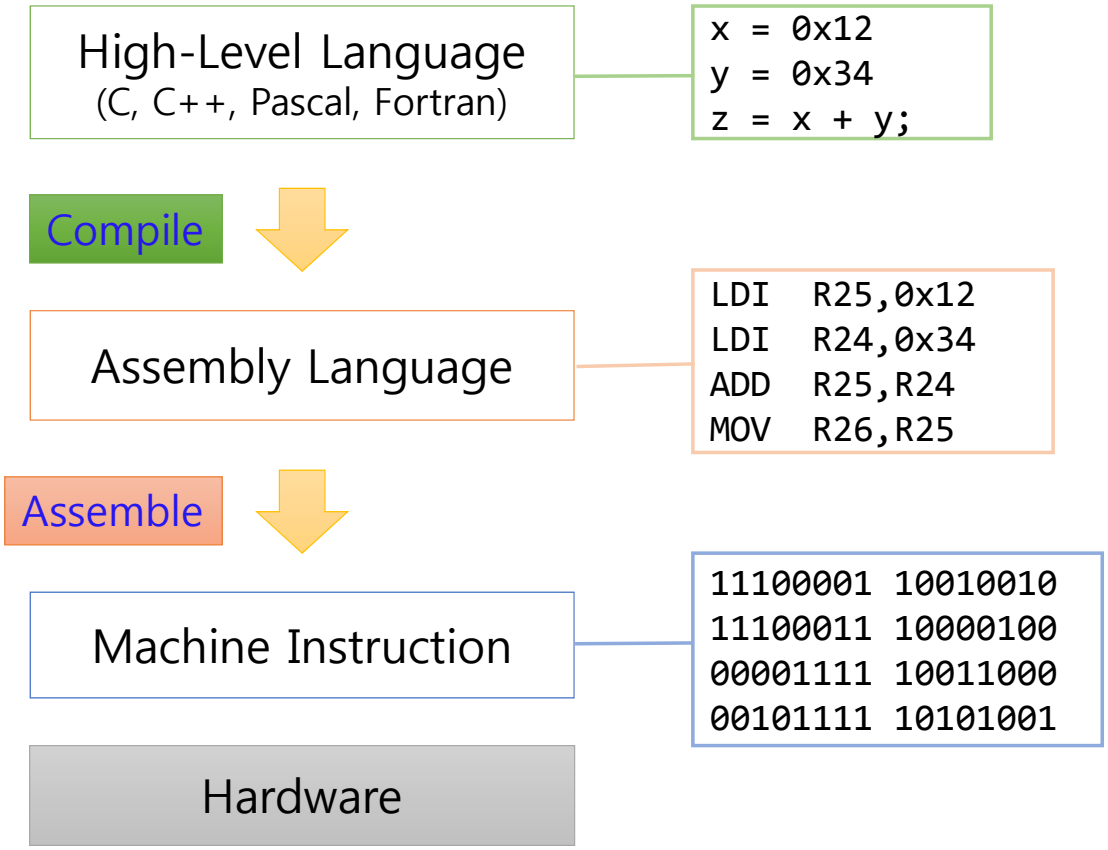
Atmega328PB
Instruction Set and Programming

Objectives

- Understanding ATmega328PB Instruction Set
- How to make a program using assembly language

Assembly Language Programming



ATmega328PB Instruction Set (1)

- The Assembler is not case sensitive.

- The operands have the following forms:

```
Rd: R0-R31 or R16-R31 (depending on instruction)
Rr: R0-R31
b: Constant (0-7), can be a constant expression
s: Constant (0-7), can be a constant expression
P: Constant (0-31/63), can be a constant expression
K: Constant (0-255), can be a constant expression
k: Constant, value range depending on instruction.
   Can be a constant expression.
q: Constant (0-63), can be a constant expression
```

ATmega328PB Instruction Set (2)

Mnemonics	Operands	Description	Operation	Flags	No. of Clock
ARITHMETIC AND LOGIC INSTRUCTIONS					
ADD	Rd, Rr	Add without Carry	Rd ← Rd + Rr	Z,C,N,V,H	1
ADC	Rd, Rr	Add with Carry	Rd ← Rd + Rr + C	Z,C,N,V,H	1
ADIW	Rd, K	Add Immediate to Word	Rd+1:Rd ← Rd+1:Rd + K	Z,C,N,V	2
SUB	Rd, Rr	Subtract without Carry	Rd ← Rd - Rr	Z,C,N,V,H	1
SUBI	Rd, K	Subtract Immediate	Rd ← Rd - K	Z,C,N,V,H	1
SBC	Rd, Rr	Subtract with Carry	Rd ← Rd - Rr - C	Z,C,N,V,H	1
SBCI	Rd, K	Subtract Immediate with Carry	Rd ← Rd - K - C	Z,C,N,V,H	1
SBIW	Rd, K	Subtract Immediate from Word	Rd+1:Rd ← Rd+1:Rd - K	Z,C,N,V	2
AND	Rd, Rr	Logical AND	Rd ← Rd ● Rr	Z,N,V	1
ANDI	Rd, K	Logical AND with Immediate	Rd ← Rd ● K	Z,N,V	1
OR	Rd, Rr	Logical OR	Rd ← Rd v Rr	Z,N,V	1
ORI	Rd, K	Logical OR with Immediate	Rd ← Rd v K	Z,N,V	1

ATmega328PB Instruction Set (3)

Mnemonics	Operands	Description	Operation	Flags	No. of Clock
ARITHMETIC AND LOGIC INSTRUCTIONS					
EOR	Rd, Rr	Exclusive OR	Rd ← Rd ⊕ Rr	Z,N,V	1
COM	Rd	One's Complement	Rd ← $FF – Rd	Z,C,N,V	1
NEG	Rd	Two's Complement	Rd ← $00 – Rd	Z,C,N,V,H	1
SBR	Rd,K	Set Bit(s) in Register	Rd ← Rd v K	Z,N,V	1
CBR	Rd,K	Clear Bit(s) in Register	Rd ← Rd ● ($FFh - K)	Z,N,V	1
INC	Rd	Increment	Rd ← Rd + 1	Z,N,V	1
DEC	Rd	Decrement	Rd ← Rd – 1	Z,N,V	1
TST	Rd	Test for Zero or Minus	Rd ← Rd ● Rd	Z,N,V	1
CLR	Rd	Clear Register	Rd ← Rd ⊕ Rd	Z,N,V	1
SER	Rd	Set Register	Rd ← $FF	None	1
MUL	Rd,Rr	Multiply Unsigned	R1, R0 ← Rd × Rr	C	2

ATmega328PB Instruction Set (4)

Mnemonics	Operands	Description	Operation	Flags	No. of Clock
BRANCH INSTRUCTIONS					
RJMP	k	Relative Jump	PC ← PC + k + 1	None	2
IJMP		Indirect Jump to (Z)	PC ← Z	None	2
JMP	k	Jump	PC ← k	None	3
RCALL	k	Relative Call Subroutine	PC ← PC + k + 1	None	3
ICALL		Indirect Call to (Z)	PC ← Z	None	3
CALL	k	Call Subroutine	PC ← k	None	4
RET		Subroutine Return	PC ← STACK	None	4
RETI		Interrupt Return	PC ← STACK	I	4
CPSE	Rd,Rr	Compare, Skip if Equal	if (Rd = Rr) PC ← PC + 2 or 3	None	1 / 2 / 3
CP	Rd,Rr	Compare	Rd - Rr	Z,C,N,V,H	1
CPC	Rd,Rr	Compare with Carry	Rd - Rr - C	Z,C,N,V,H	1
CPI	Rd,K	Compare with Immediate	Rd - K	Z,C,N,V,H	1
SBRC	Rr, b	Skip if Bit in Register Cleared	if (Rr(b)=0) PC ← PC + 2 or 3	None	1 / 2 / 3
SBRS	Rr, b	Skip if Bit in Register Set	if (Rr(b)=1) PC ← PC + 2 or 3	None	1 / 2 / 3
SBIC	P, b	Skip if Bit in I/O Register Cleared	if(I/O(P,b)=0) PC ← PC + 2 or 3	None	1 / 2 / 3
SBIS	P, b	Skip if Bit in I/O Register Set	if(I/O(P,b)=1) PC ← PC + 2 or 3	None	1 / 2 / 3
BRBS	s, k	Branch if Status Flag Set	if (SREG(s) = 1) then PC ← PC + k + 1	None	1 / 2
BRBC	s, k	Branch if Status Flag Cleared	if (SREG(s) = 0) then PC ← PC + k + 1	None	1 / 2

ATmega328PB Instruction Set (5)

Mnemonics	Operands	Description	Operation	Flags	No. of Clock
BRANCH INSTRUCTIONS					
BREQ	k	Branch if Equal	if (Z = 1) then PC ← PC + k + 1	None	1 / 2
BRNE	k	Branch if Not Equal	if (Z = 0) then PC ← PC + k + 1	None	1 / 2
BRCS	k	Branch if Carry Set	if (C = 1) then PC ← PC + k + 1	None	1 / 2
BRCC	k	Branch if Carry Cleared	if (C = 0) then PC ← PC + k + 1	None	1 / 2
BRSH	k	Branch if Same or Higher	if (C = 0) then PC ← PC + k + 1	None	1 / 2
BRLO	k	Branch if Lower	if (C = 1) then PC ← PC + k + 1	None	1 / 2
BRMI	k	Branch if Minus	if (N = 1) then PC ← PC + k + 1	None	1 / 2
BRPL	k	Branch if Plus	if (N = 0) then PC ← PC + k + 1	None	1 / 2
BRGE	k	Branch if Greater or Equal, Signed	if (N ⊕ V= 0) then PC ← PC+ k + 1	None	1 / 2
BRLT	k	Branch if Less Than, Signed	if (N ⊕ V= 1) then PC ← PC + k + 1	None	1 / 2
BRHS	k	Branch if Half Carry Flag Set	if (H = 1) then PC ← PC + k + 1	None	1 / 2
BRHC	k	Branch if Half Carry Flag Cleared	if (H = 0) then PC ← PC + k + 1	None	1 / 2
BRTS	k	Branch if T Flag Set	if (T = 1) then PC ← PC + k + 1	None	1 / 2
BRTC	k	Branch if T Flag Cleared	if (T = 0) then PC ← PC + k + 1	None	1 / 2
BRVS	k	Branch if Overflow Flag is Set	if (V = 1) then PC ← PC + k + 1	None	1 / 2
BRVC	k	Branch if Overflow Flag is Cleared	if (V = 0) then PC ← PC + k + 1	None	1 / 2
BRIE	k	Branch if Interrupt Enabled	if (I = 1) then PC ← PC + k + 1	None	1 / 2
BRID	k	Branch if Interrupt Disabled	if (I = 0) then PC ← PC + k + 1	None	1 / 2

ATmega328PB Instruction Set (6)

Mnemonics	Operands	Description	Operation	Flags	No. of Clock
DATA TRANSFER INSTRUCTIONS					
MOV	Rd, Rr	Copy Register	Rd ← Rr	None	1
LDI	Rd, K	Load Immediate	Rd ← K	None	1
LDS	Rd, k	Load Direct from SRAM	Rd ← (k)	None	3
LD	Rd, X	Load Indirect	Rd ← (X)	None	2
LD	Rd, X+	Load Indirect and Post-Increment	Rd ← (X), X ← X + 1	None	2
LD	Rd, -X	Load Indirect and Pre-Decrement	X ← X - 1, Rd ← (X)	None	2
LD	Rd, Y	Load Indirect	Rd ← (Y)	None	2
LD	Rd, Y+	Load Indirect and Post-Increment	Rd ← (Y), Y ← Y + 1	None	2
LD	Rd, -Y	Load Indirect and Pre-Decrement	Y ← Y - 1, Rd ← (Y)	None	2
LDD	Rd,Y+q	Load Indirect with Displacement	Rd ← (Y + q)	None	2
LD	Rd, Z	Load Indirect	Rd ← (Z)	None	2
LD	Rd, Z+	Load Indirect and Post-Increment	Rd ← (Z), Z ← Z+1	None	2
LD	Rd, -Z	Load Indirect and Pre-Decrement	Z ← Z - 1, Rd ← (Z)	None	2
LDD	Rd, Z+q	Load Indirect with Displacement	Rd ← (Z + q)	None	2
STS	k, Rr	Store Direct to SRAM	(k) ← Rr	None	3
ST	X, Rr	Store Indirect	(X) ← Rr	None	2

ATmega328PB Instruction Set (7)

Mnemonics	Operands	Description	Operation	Flags	No. of Clock
DATA TRANSFER INSTRUCTIONS					
ST	X+, Rr	Store Indirect and Post-Increment	(X) ← Rr, X ← X + 1	None	2
ST	-X, Rr	Store Indirect and Pre-Decrement	X ← X - 1, (X) ← Rr	None	2
ST	Y, Rr	Store Indirect	(Y) ← Rr	None	2
ST	Y+, Rr	Store Indirect and Post-Increment	(Y) ← Rr, Y ← Y + 1	None	2
ST	-Y, Rr	Store Indirect and Pre-Decrement	Y ← Y - 1, (Y) ← Rr	None	2
STD	Y+q,Rr	Store Indirect with Displacement	(Y + q) ← Rr	None	2
ST	Z, Rr	Store Indirect	(Z) ← Rr	None	2
ST	Z+, Rr	Store Indirect and Post-Increment	(Z) ← Rr, Z ← Z + 1	None	2
ST	-Z, Rr	Store Indirect and Pre-Decrement	Z ← Z - 1, (Z) ← Rr	None	2
STD	Z+q,Rr	Store Indirect with Displacement	(Z + q) ← Rr	None	2
LPM		Load Program Memory	R0 ← (Z)	None	3
IN	Rd, P	In Port	Rd ← P	None	1
OUT	P, Rr	Out Port	P ← Rr	None	1
PUSH	Rr	Push Register on Stack	STACK ← Rr	None	2
POP	Rd	Pop Register from Stack	Rd ← STACK	None	2

ATmega328PB Instruction Set (8)

Mnemonics	Operands	Description	Operation	Flags	No. of Clock
BIT AND BIT-TEST INSTRUCTIONS					
LSL	Rd	Logical Shift Left	Rd(n+1) ← Rd(n), Rd(0) ← 0, C ← Rd(7)	Z,C,N,V,H	1
LSR	Rd	Logical Shift Right	Rd(n) ← Rd(n+1), Rd(7) ← 0, C ← Rd(0)	Z,C,N,V	1
ROL	Rd	Rotate Left Through Carry	Rd(0) ← C, Rd(n+1) ← Rd(n), C ← Rd(7)	Z,C,N,V,H	1
ROR	Rd	Rotate Right Through Carry	Rd(7) ← C, Rd(n) ← Rd(n+1), C ← Rd(0)	Z,C,N,V	1
ASR	Rd	Arithmetic Shift Right	Rd(n) ← Rd(n+1), n=0..6	Z,C,N,V	1
SWAP	Rd	Swap Nibbles	Rd(3..0) ↔ Rd(7..4)	None	1
BSET	s	Flag Set	SREG(s) ← 1	SREG(s)	1
BCLR	s	Flag Clear	SREG(s) ← 0	SREG(s)	1
SBI	P, b	Set Bit in I/O Register	I/O(P, b) ← 1	None	2
CBI	P, b	Clear Bit in I/O Register	I/O(P, b) ← 0	None	2
BST	Rr, b	Bit Store from Register to T	T ← Rr(b)	T	1
BLD	Rd, b	Bit load from T to Register	Rd(b) ← T	None	1
SEC		Set Carry	C ← 1	C	1
CLC		Clear Carry	C ← 0	C	1
SEN		Set Negative Flag	N ← 1	N	1
CLN		Clear Negative Flag	N ← 0	N	1

ATmega328PB Instruction Set (9)

Mnemonics	Operands	Description	Operation	Flags	No. of Clock
BIT AND BIT-TEST INSTRUCTIONS					
CLN		Clear Negative Flag	N ← 0	N	1
SEZ		Set Zero Flag	Z ← 1	Z	1
CLZ		Clear Zero Flag	Z ← 0	Z	1
SEI		Global Interrupt Enable	I ← 1	I	1
CLI		Global Interrupt Disable	I ← 0	I	1
SES		Set Signed Test Flag	S ← 1	S	1
CLS		Clear Signed Test Flag	S ← 0	S	1
SEV		Set Two's Complement Overflow	V ← 1	V	1
CLV		Clear Two's Complement Overflow	V ← 0	V	1
SET		Set T in SREG	T ← 1	T	1
CLT		Clear T in SREG	T ← 0	T	1
SEH		Set Half Carry Flag in SREG	H ← 1	H	1
CLH		Clear Half Carry Flag in SREG	H ← 0	H	1
NOP		No Operation		None	1
SLEEP		Sleep		None	1
WDR		Watchdog Reset		None	1

Assembly Language Statement Format

- An assembly language statement line may take one of the four following forms:
  - `[label:] directive [operands] [Comment]`
  - `[label:] instruction [operands] [Comment]`
  - Comment
  - Empty line
- A comment has the following form:

  `; [Text]`
- Items placed in braces are optional.
- The text between the comment-delimiter (;) and the end of line (EOL) is ignored by the Assembler.
- Examples:

```
label: .EQU var1=100     ; Set var1 to 100 (Directive)
       .EQU var2=200     ; Set var2 to 200
test:  rjmp test         ; Infinite loop (Instruction)
; Pure comment line
                         ; Another comment line
```

Note: There are no restrictions with respect to column placement of labels, directives, comments or instructions.

Assembler Directives (1)

- The directives are not translated directly into opcodes.
- They are used to adjust the location of the program in memory, define macros, initialize memory and so on.

Assembler Directives (2)

Directive	Description
BYTE	Reserve byte to a variable
CSEG	Code Segment
DB	Define constant byte(s)
DEF	Define a symbolic name on a register
DEVICE	Define which device to assemble for
DSEG	Data Segment
DW	Define constant word(s)
ENDMACRO	End macro
EQU	Set a symbol equal to an expression

Directive	Description
ESEG	EEPROM Segment
EXIT	Exit from file
INCLUDE	Read source from another file
LIST	Turn list file generation on
LISTMAC	Turn macro expansion on
MACRO	Begin macro
NOLIST	Turn list file generation off
ORG	Set program origin
SET	Set a symbol to an expression

Note: All directives must be preceded by a period.

Assembler Directives (3)

CSEG – Code Segment
The CSEG directive defines the start of a Code Segment. An Assembler file can consist of several Code Segments, which are concatenated into one Code Segment when assembled. The BYTE directive can not be used within a Code Segment. The default segment type is Code. The Code Segments have their own location counter which is a word counter. The ORG directive (see description later in this document) can be used to place code and constants at specific locations in the Program memory. The directive does not take any parameters.
`.CSEG`

```
        .DSEG          ; Start data segment
vartab: .BYTE 4        ; Reserve 4 bytes in SRAM
        .CSEG          ; Start code segment
const:  .DW   2        ; Write 0x0002 in program memory
        mov   r1,r0    ; Do something
```

Assembler Directives (4)

DSEG – Data Segment

The DSEG directive defines the start of a Data Segment. An Assembler file can consist of several Data Segments, which are concatenated into one Data Segment when assembled. A Data Segment will normally only consist of BYTE directives (and labels). The Data Segments have their own location counter which is a byte counter. The ORG directive (see description later in this document) can be used to place the variables at specific locations in the SRAM. The directive does not take any parameters.

```
        .DSEG
```

```
        .DSEG
var1:   .BYTE   1           ; reserve 1 byte to var1
table:  .BYTE   6           ; reserve 6 bytes to table
        .CSEG
        ldi r30,low(var1)   ; Load Z register low
        ldi r31,high(var1)  ; Load Z register high
        ld  r1,Z            ; Load var1 into r1
```

Assembler Directives (5)

ESEG – EEPROM Segment

The ESEG directive defines the start of an EEPROM Segment. An Assembler file can consist of several EEPROM Segments, which are concatenated into one EEPROM Segment when assembled. The BYTE directive can not be used within an EEPROM Segment. The EEPROM Segments have their own location counter which is a byte counter. The ORG directive (see description later in this document) can be used to place constants at specific locations in the EEPROM memory. The directive does not take any parameters.

```
        .ESEG
```

```
        .DSEG             ; Start data segment
vartab: .BYTE  4          ; Reserve 4 bytes in SRAM
        .ESEG
eevar:  .DW    0xff0f     ; Initialize one word in EEPROM
        .CSEG             ; Start code segment
const:  .DW    2          ; Write 0x0002 in prog.mem.
        mov    r1,r0      ; Do something
```

Assembler Directives (6)

BYTE – Reserve bytes to a variable

The BYTE directive reserves memory resources in the SRAM. In order to be able to refer to the reserved location, the BYTE directive should be preceded by a label. The directive takes one parameter, which is the number of bytes to reserve. The directive can only be used within a Data Segment (see directives CSEG, DSEG and ESEG).
Note that a parameter must be given. The allocated bytes are not initialized.

LABEL: .BYTE expression

```
        .DSEG
var1:   .BYTE   1           ; reserve 1 byte to var1
table:  .BYTE   6           ; reserve 6 bytes to table
        .CSEG
        ldi r30,low(var1)   ; Load Z register low
        ldi r31,high(var1)  ; Load Z register high
        ld r1,Z             ; Load VAR1 into register 1
```

Assembler Directives (7)

DB – Define constant byte(s) in program memory or EEPROM memory

The DB directive reserves memory resources in the program memory or the EEPROM memory. In order to be able to refer to the reserved locations, the DB directive should be preceded by a label.
The DB directive takes a list of expressions, and must contain at least one expression. The DB directive must be placed in a Code Segment or an EEPROM Segment.
The expression list is a sequence of expressions, delimited by commas. Each expression must evaluate to a number between -128 and 255. If the expression evaluates to a negative number, the 8 bits two's complement of the number will be placed in the program memory or EEPROM memory location.
If the DB directive is used in a Code Segment and the expression_list contains more than one expression, the expressions are packed so that two bytes are placed in each program memory word. If the expression_list contains an odd number of expressions, the last expression will be placed in a program memory word of its own, even if the next line in the assembly code contains a DB directive.

```
LABEL:   .DB   expression_list
```

```
         .CSEG
consts:  .DB   0, 255, 0b01010101, -128, 0xaa
         .ESEG
eeconst: .DB   0xff
```

Assembler Directives (8)

DW – Define constant word(s) in program memory or E2PROM memory
The DW directive reserves memory resources in the program memory or EEPROM memory. In order to be able to refer to the reserved locations, the DW directive should be preceded by a label. The DW directive takes a list of expressions, and must contain at least one expression. The DW directive must be placed in a Code Segment or an EEPROM Segment. The expression list is a sequence of expressions, delimited by commas. Each expression must evaluate to a number between -32768 and 65535. If the expression evaluates to a negative number, the 16 bits two's complement of the number will be placed in the program memory location.
LABEL: .DW expression_list

```
        .CSEG
varlist: .DW   0,0xffff,0b1001110001010101,-32768,65535
        .ESEG
eevar:  .DW    0xffff
```

Assembler Directives (9)

DEF – Set a symbolic name on a register

The DEF directive allows the registers to be referred to through symbols. A defined symbol can be used in the rest of the program to refer to the register it is assigned to. A register can have several symbolic names attached to it. A symbol can be redefined later in the program.

LABEL: *.DEF Symbol = Register*

```
        .DEF temp = R16
        .DEF ior = R0
        .CSEG
        ldi temp,0xf0      ; Load 0xf0 into temp register
        in  ior,0x3f       ; Read SREG into ior register
        eor temp,ior       ; Exclusive or temp and ior
```

Assembler Directives (10)

EQU – Set a symbol equal to an expression
The EQU directive assigns a value to a label. This label can then be used in later expressions. A label assigned to a value by the EQU directive is a constant and can not be changed or redefined.
`.EQU  label = expression`

`.EQU  io_offset = 0x23`
`.EQU  porta = io_offset + 2`
`.CSEG  ; Start code segment`
`clr  r2  ; Clear register 2`
`out porta,r2  ; Write to Port A`

Assembler Directives (11)

SET – Set a symbol equal to an expression
The SET directive assigns a value to a label. This label can then be used in later expressions. A label assigned to a value by the SET directive can be changed later in the program.
`.SET  label = expression`

```
.SET   io_offset = 0x23
.SET   porta = io_offset + 2
.CSEG              ; Start code segment
clr  r2            ; Clear register 2
out porta,r2       ; Write to Port A
```

Assembler Directives (12)

ORG – Set program origin

The ORG directive sets the location counter to an absolute value. The value to set is given as a parameter.
If an ORG directive is given within a Data Segment, then it is the SRAM location counter which is set.
If the directive is given within a Code Segment, then it is the Program memory counter which is set.
If the directive is given within an EEPROM Segment, then it is the EEPROM location counter which is set.
If the directive is preceded by a label (on the same source code line), the label will be given the value of the parameter.
The default values of the Code and EEPROM location counters are zero, whereas the default value of the SRAM location counter is 32 (due to the registers occupying addresses 0-31) when the assembling is started. Note that the EEPROM and SRAM location counters count bytes whereas the Program memory location counter counts words.

```
        .ORG    expression
```

```
        .DSEG           ; Start data segment
        .ORG    0x67    ; Set SRAM address to hex 67
rvar:   .BYTE   1       ; Reserve a byte at SRAM address 67H
        .ESEG           ; Start EEPROM Segment
        .ORG    0x20    ; Set EEPROM location counter
eevar:  .DW     0xfeff  ; Initialize one word
        .CSEG
        .ORG    0x10    ; Set Program Counter to hex 10
        mov     r0,r1   ; Do something
```

Assembler Directives (13)

INCLUDE – Include another file

The INCLUDE directive tells the Assembler to start reading from a specified file. The Assembler then assembles the specified file until end of file (EOF) or an EXIT directive is encountered. An included file may itself contain INCLUDE directives.

```
.INCLUDE   "filename"
```

```
.EQU sreg   = 0x3f       ; Status register
.EQU sphigh = 0x3e       ; Stack pointer high
.EQU splow  = 0x3d       ; Stack pointer low
                         ; incdemo.asm
.INCLUDE  "iodefs.asm"   ; Include I/O definitions
in r0,sreg               ; Read status register
```

Expressions (1)

Expressions can consist of operands, operators and functions. All expressions are internally 32 bits.

Operands

The following operands can be used:

- User defined labels which are given the value of the location counter at the place they appear.
- User defined variables defined by the SET directive
- User defined constants defined by the EQU directive
- Integer constants: constants can be given in several formats, including,
  - `Decimal (default): 10, 255`
  - `Hexadecimal (two notations): 0x0a, $0a, 0xff, $ff`
  - `Binary: 0b00001010, 0b11111111`
- PC - the current value of the Program memory location counter

Expressions (2)

Functions

The following functions are defined:

- LOW(expression) returns the low byte of an expression
- HIGH(expression) returns the second byte of an expression
- BYTE2(expression) is the same function as HIGH
- BYTE3(expression) returns the third byte of an expression
- BYTE4(expression) returns the fourth byte of an expression
- LWRD(expression) returns bits 0-15 of an expression
- HWRD(expression) returns bits 16-31 of an expression
- PAGE(expression) returns bits 16-21 of an expression
- EXP2(expression) returns 2^expression
- LOG2(expression) returns the integer part of log2(expression)

Expressions (3): Operators

Logical NOT

Symbol: !
Description: Unary operator which returns 1 if the expression was zero, and returns 0 if the expression was nonzero
Precedence: 14
Example:

```
ldi r16,!0xf0              ; Load r16 with 0x00
```

Bitwise NOT

Symbol: ~
Description: Unary operator which returns the input expression with all bits inverted
Precedence: 14
Example:

```
ldi r16,~0xf0              ; Load r16 with 0x0f
```

Expressions (4): Operators

Unary Minus

Symbol: -
Description: Unary operator which returns the arithmetic negation of an expression
Precedence: 14
Example:

```
ldi r16,-2                ; Load -2(0xfe) in r16
```

Multiplication

Symbol: *
Description: Binary operator which returns the product of two expressions
Precedence: 13
Example:

```
ldi r30,label*2           ; Load r30 with label*2
```

Expressions (5): Operators

Division

Symbol: /
Description: Unary operator which returns the arithmetic negation of an expression
Precedence: 13
Example:

```
ldi r30,label/2          ; Load r30 with label/2
```

Addition

Symbol: +
Description: Binary operator which returns the sum of two expressions
Precedence: 12
Example:

```
ldi r30,c1+c2            ; Load r30 with c1+c2
```

Expressions (6): Operators

Subtraction

Symbol: -
Description: Binary operator which returns the left expression minus the right expression
Precedence: 12
Example:

```
ldi r17,c1-c2              ; Load r17 with c1-c2
```

Shift left

Symbol: <<
Description: Binary operator which returns the left expression shifted left a number of times given by the right expression
Precedence: 11
Example:

```
ldi r17,1<<5       ; Load r17 with 1 shifted left 5 times
```

Expressions (7): Operators

Shift right

Symbol: >>
Description: Binary operator which returns the left expression shifted right a number of times given by the right expression
Precedence: 11
Example:

```
ldi r17,c1>>c2          ; Load r17 with c1 shifted right c2 times
```

Bitwise AND

Symbol: &
Description: Binary operator which returns the bitwise And between two expressions
Precedence: 8
Example:

```
ldi r18,High(c1&c2)     ; Load r18 with an expression
```

Expressions (8): Operators

Bitwise XOR

Symbol: ^
Description: Binary operator which returns the bitwise Exclusive Or between two expressions
Precedence: 7
Example:

```
ldi r18,Low(c1^c2)                ; Load r18 with an expression
```

Bitwise OR

Symbol: |
Description: Binary operator which returns the bitwise Or between two expressions
Precedence: 6
Example:

```
ldi r18,Low(c1|c2)                ; Load r18 with an expression
```

Recommendation for the use of registers

- Define names for registers with the .DEF directive, never use them with their direct name Rx.
- If you need pointer access reserve R26 to R31 for that purpose.
- A 16-bit-counter is best located in R25:R24.
- If you need to read from the program memory reserve Z (R31:R30) and R0 for that purpose.
- If you need to access to single bits within certain registers, use R16 to R23 for that purpose.
- Registers necessary for math are best placed to R1 to R15.
- If you have more than enough registers available, place all your variables in registers.
- If you get short in registers, place as many variables as necessary to SRAM.

Assembly Language Programming Example (1)

- Define names for registers with the .DEF directive, never use them with their direct name Rx.
- If you need pointer access reserve R26 to R31 for that purpose.
- A 16-bit-counter is best located in R25:R24.
- If you need to read from the program memory reserve Z (R31:R30) and R0 for that purpose.
- If you need to access to single bits within certain registers, use R16 to R23 for that purpose.
- Registers necessary for math are best placed to R1 to R15.
- If you have more than enough registers available, place all your variables in registers.
- If you get short in registers, place as many variables as necessary to SRAM.

Mixed Language Programming (1)

Example: A C language function calls an assembly language function.

```c
/* main.c */

#include <avr/io.h>

extern void asm_func(uint8_t val);

int main(void)
{
    DDRB = 0xFF;
    asm_func(3);
}
```

```c
/* asm_files.s */

#define __SFR_OFFSET 0
#include <avr/io.h>

.global  asm_func
.section .text

asm_func:
        OUT   PORTB,R24
        RET
```

#define __SFR_OFFSET 0
To use these addresses in in/out instructions, you must subtract 0x20 from them.

Mixed Language Programming (2)

Passing Parameters between C and Assembly Functions

- Parameters are passed via R25:R8 (R25 to R8).
- Parameters are passed left to right.

```
uint8_t function(uint8_t i, uint8_t j);
```

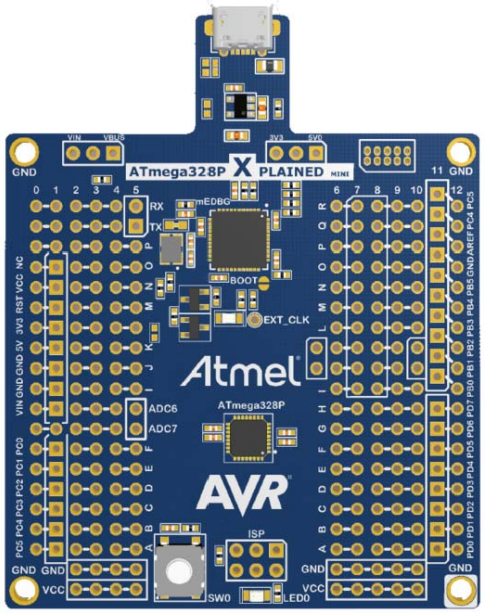
- In this example,
  - i would be stored in R25:24 (with the actual 8-bit value stored in R24).
  - j would be stored in R23:22 (with the actual 8-bit value stored in R22).
- If the parameters passed require more memory than is available in the registers R25:R8, then the stack is used to pass additional parameters.
- Return values are placed in registers beginning at R25.
  - An 8-bit value gets returned in R24.
  - An 16-bit value gets returned in R25:R24.
  - An 32-bit value gets returned in R25:R22.
  - An 64-bit value gets returned in R25:R18.

Application Development Tools (1)

- Atmel Studio - Software
  - ✓ Integrated Development Environment (IDE)
  - ✓ Edit, compile, link, debug
  - ✓ Simulation and FLASH programming
  - ✓ Includes GNU C/C++ compiler

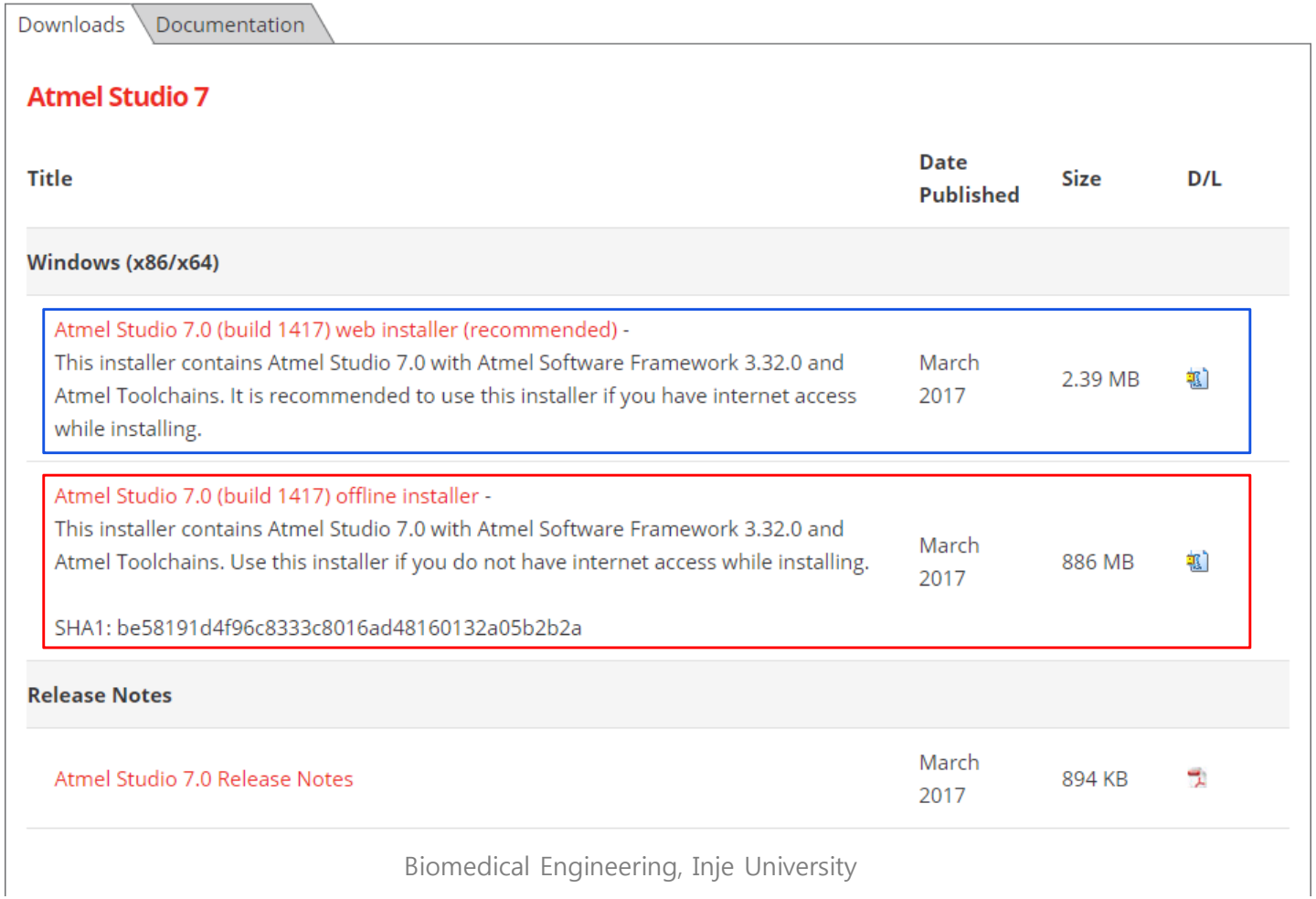
Application Development Tools (2)

- ATmega328P(B) Xplained Mini Evaluation Kit - Hardware
  - ✓ On-board debugger with full source-level debugging support in Atmel Studio.
  - ✓ Auto-ID for board identification in Atmel Studio 7
  - ✓ Access to all signals on target MCU.
  - ✓ One yellow user LED / One mechanical user pushbutton
  - ✓ Virtual COM port (CDC)
  - ✓ 16MHz target clock
  - ✓ USB powered
  - ✓ Arduino shield compatible foot prints
  - ✓ Supported with application examples published on Atmel Spaces



Download and Install Atmel Studio 7

http://www.microchip.com/development-tools/atmel-studio-7

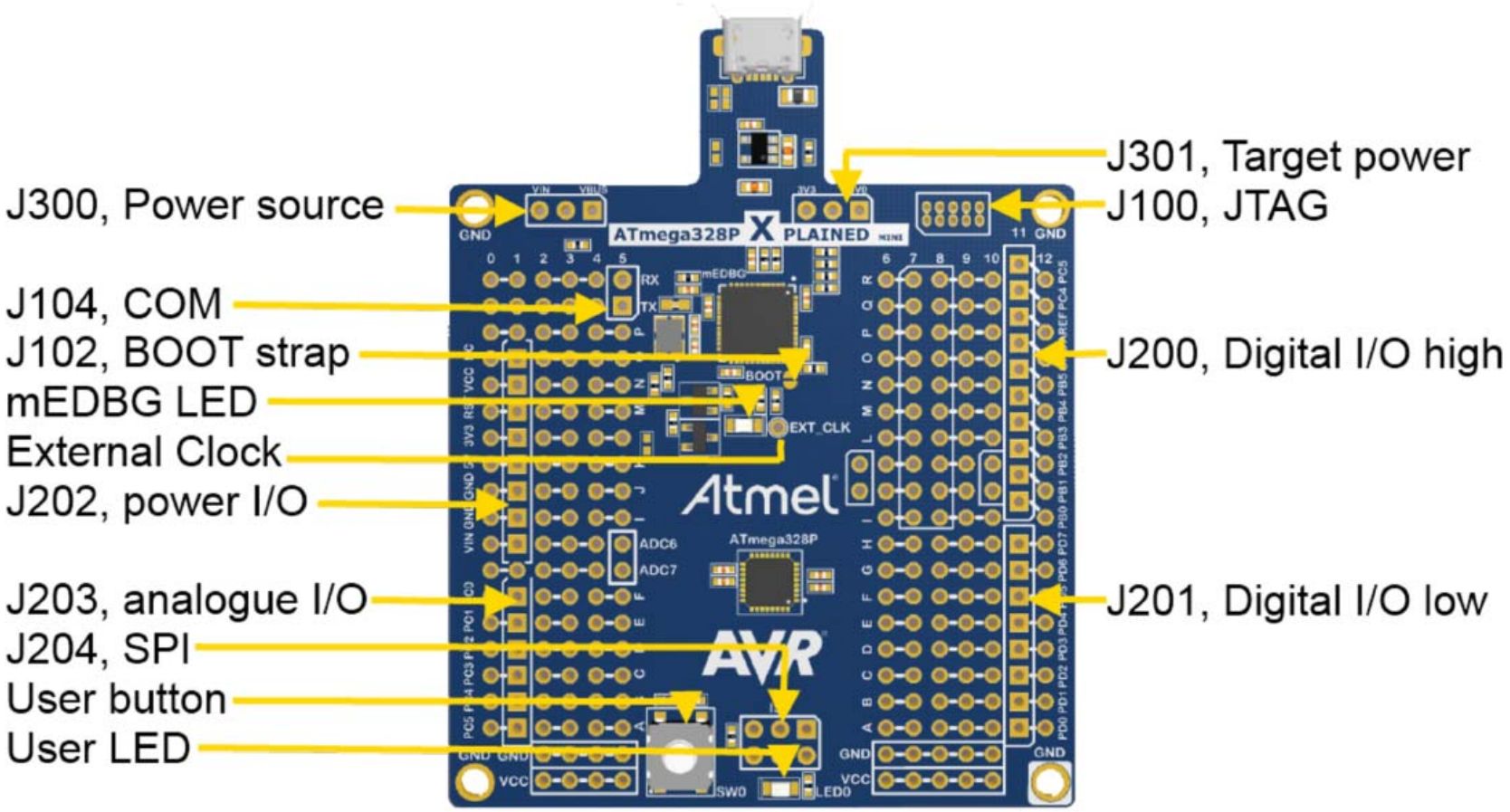


Downloads | Documentation

Atmel Studio 7

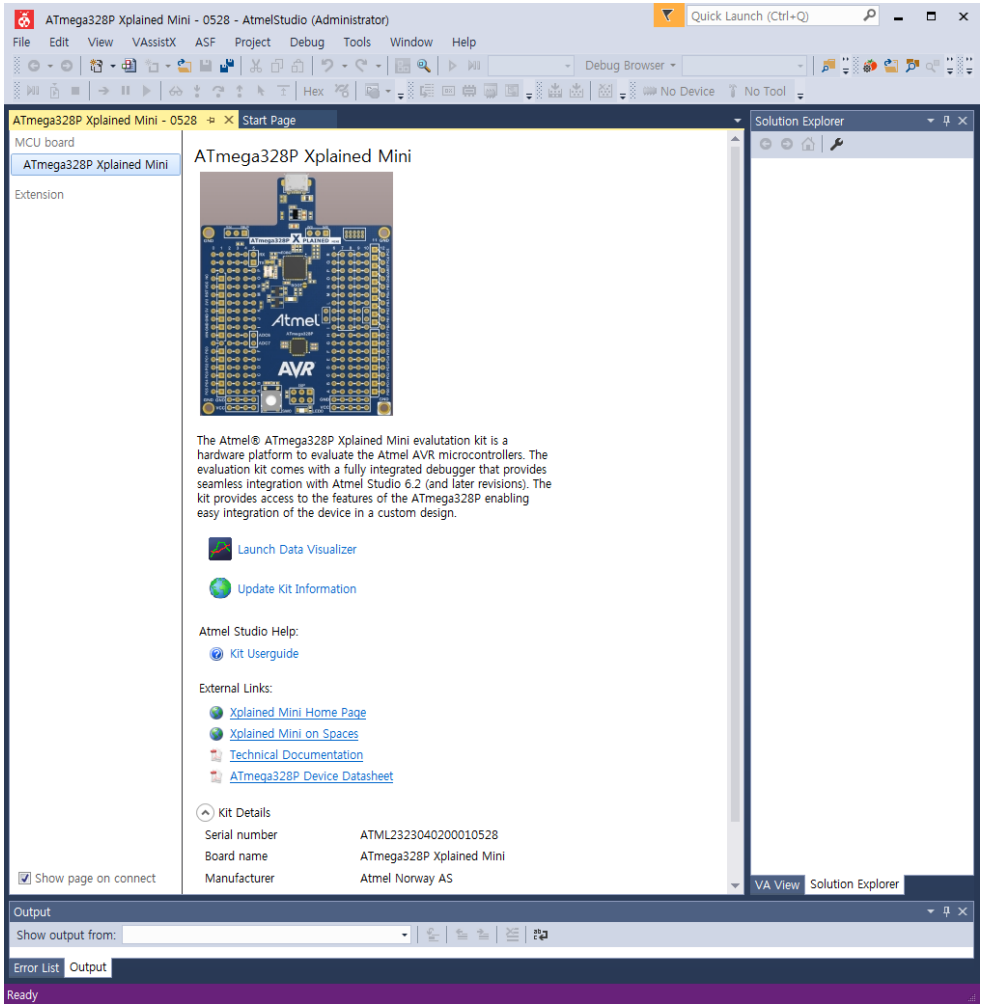
Title	Date Published	Size	D/L
Windows (x86/x64)			
Atmel Studio 7.0 (build 1417) web installer (recommended) - This installer contains Atmel Studio 7.0 with Atmel Software Framework 3.32.0 and Atmel Toolchains. It is recommended to use this installer if you have internet access while installing.	March 2017	2.39 MB	
Atmel Studio 7.0 (build 1417) offline installer - This installer contains Atmel Studio 7.0 with Atmel Software Framework 3.32.0 and Atmel Toolchains. Use this installer if you do not have internet access while installing. SHA1: be58191d4f96c8333c8016ad48160132a05b2b2a	March 2017	886 MB	
Release Notes			
Atmel Studio 7.0 Release Notes	March 2017	894 KB	

ATmega328P(B) Xplained Mini Evaluation Kit



Create an AVR Application in Assembly Language (1)

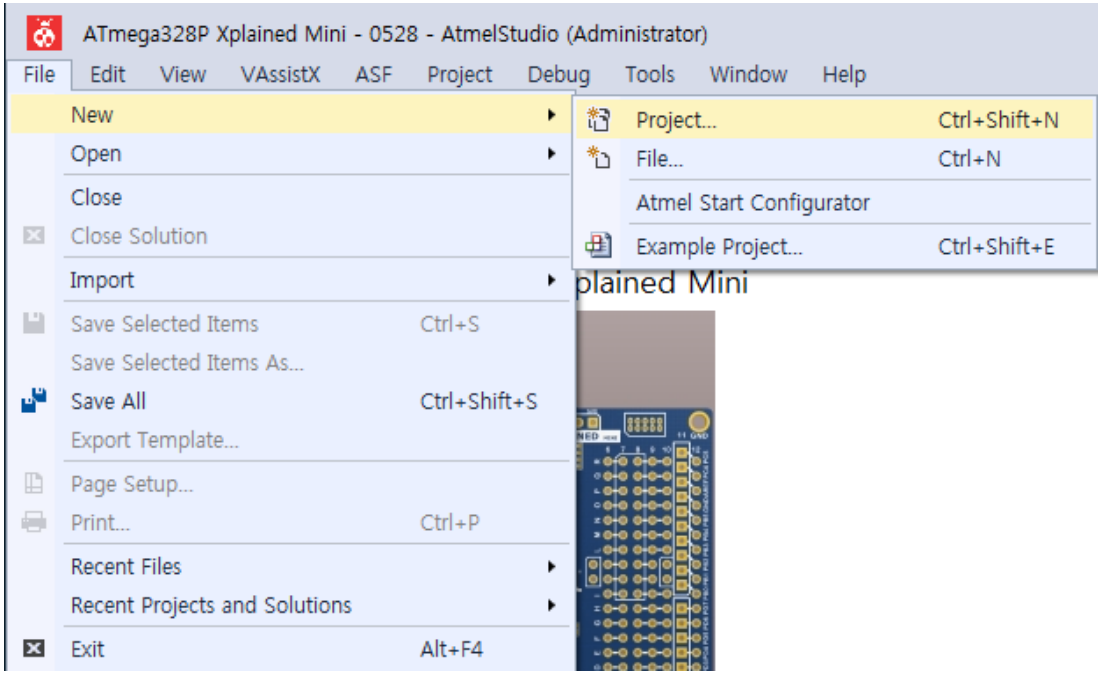
1. Run AVR Studio 7.
2. Connect ATmega328P(B) Xplained Mini to PC via USB Micro-B cable.
3. Image of the detected ATmega328P Xplained Mini is displayed with related information.



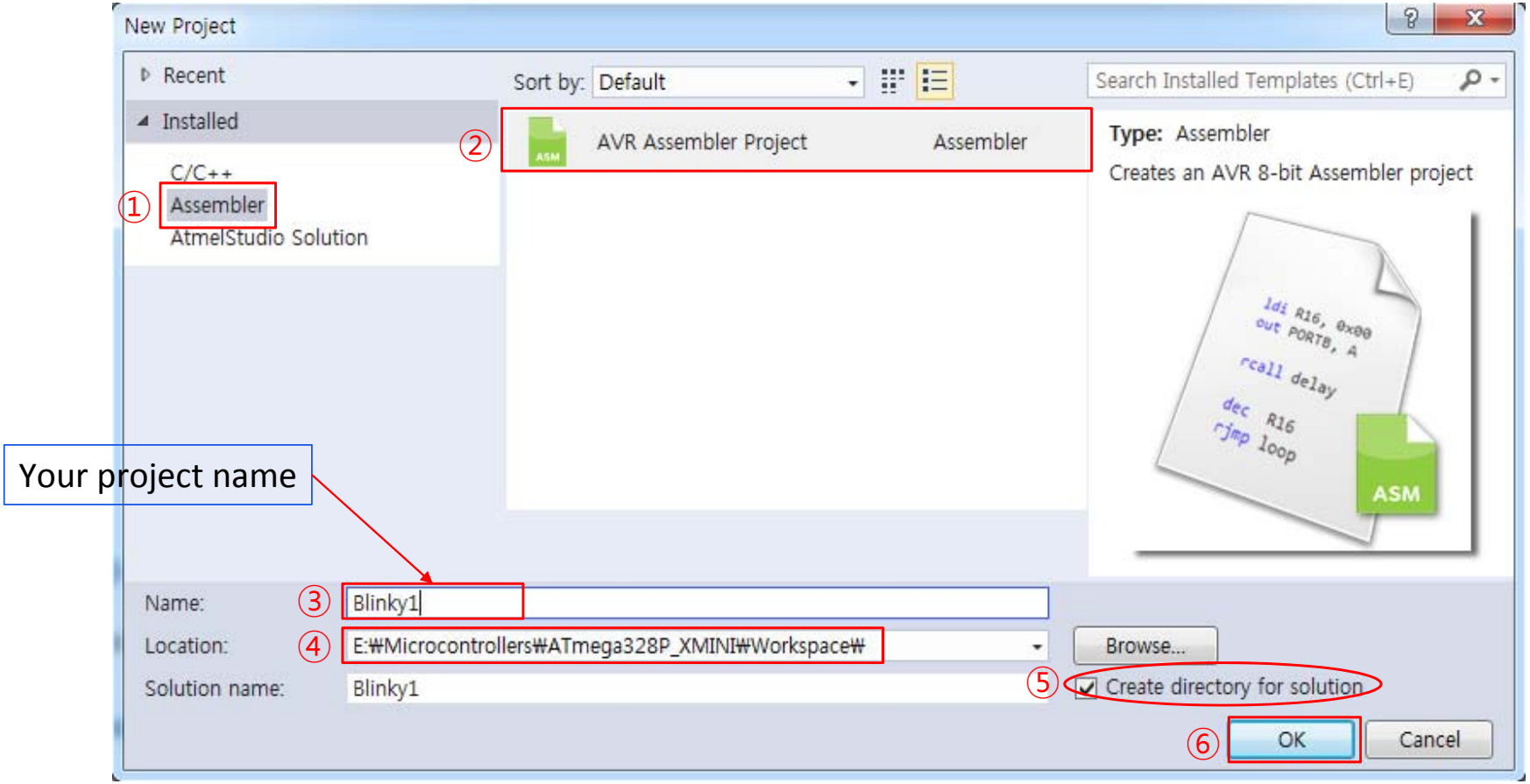
Create an AVR Application in Assembly Language (2)

1. Select

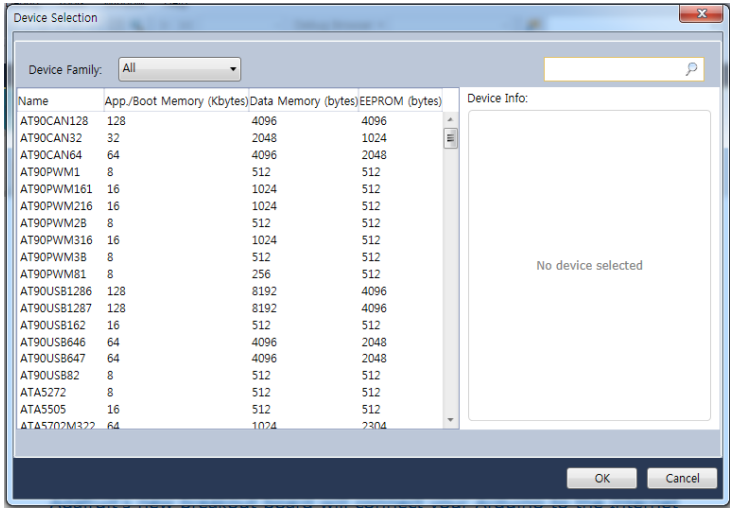
   File – New – Project…

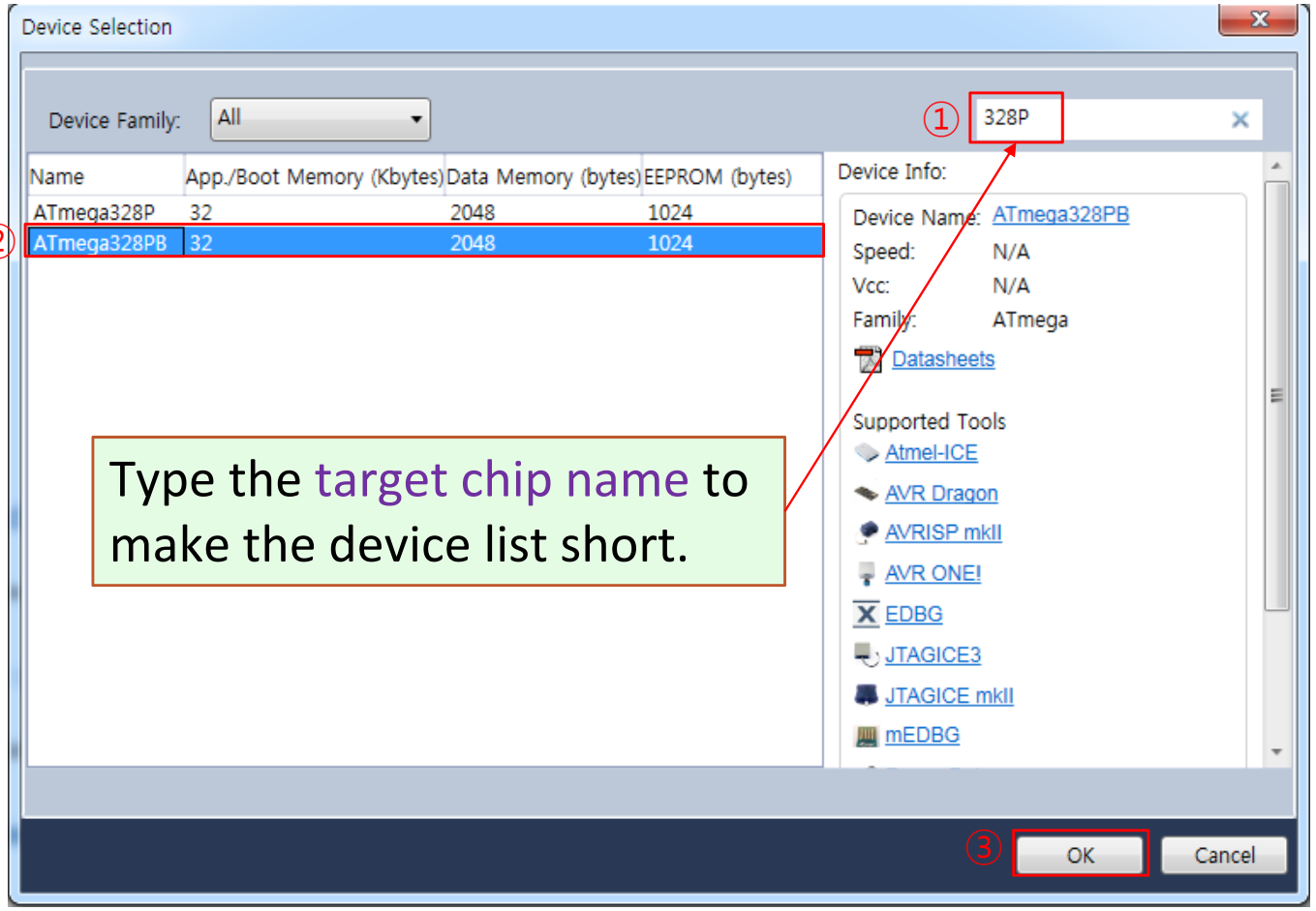


Create an AVR Application in Assembly Language (3)



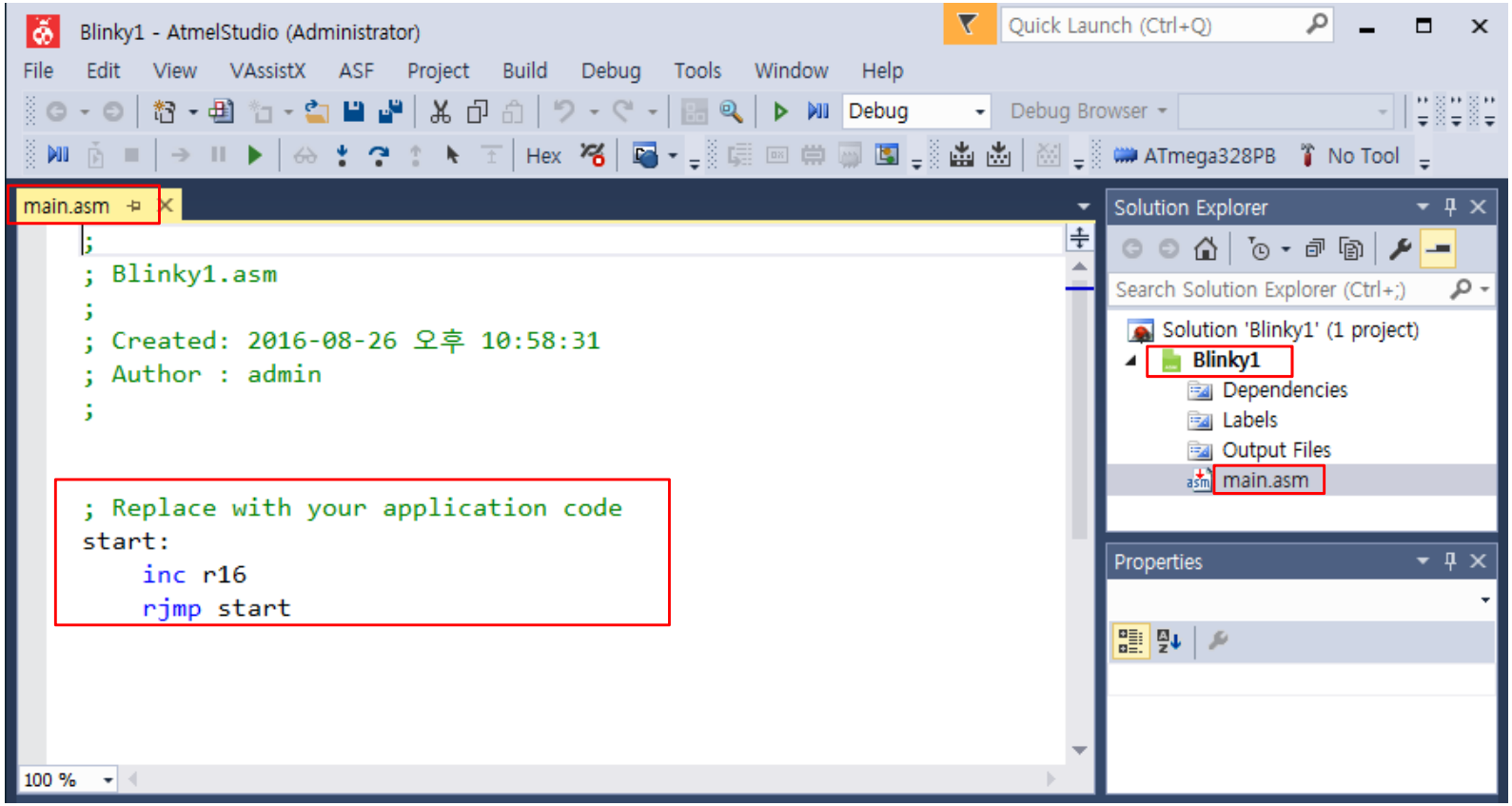
Create an AVR Application in Assembly Language (4)





Create an AVR Application in Assembly Language (5)

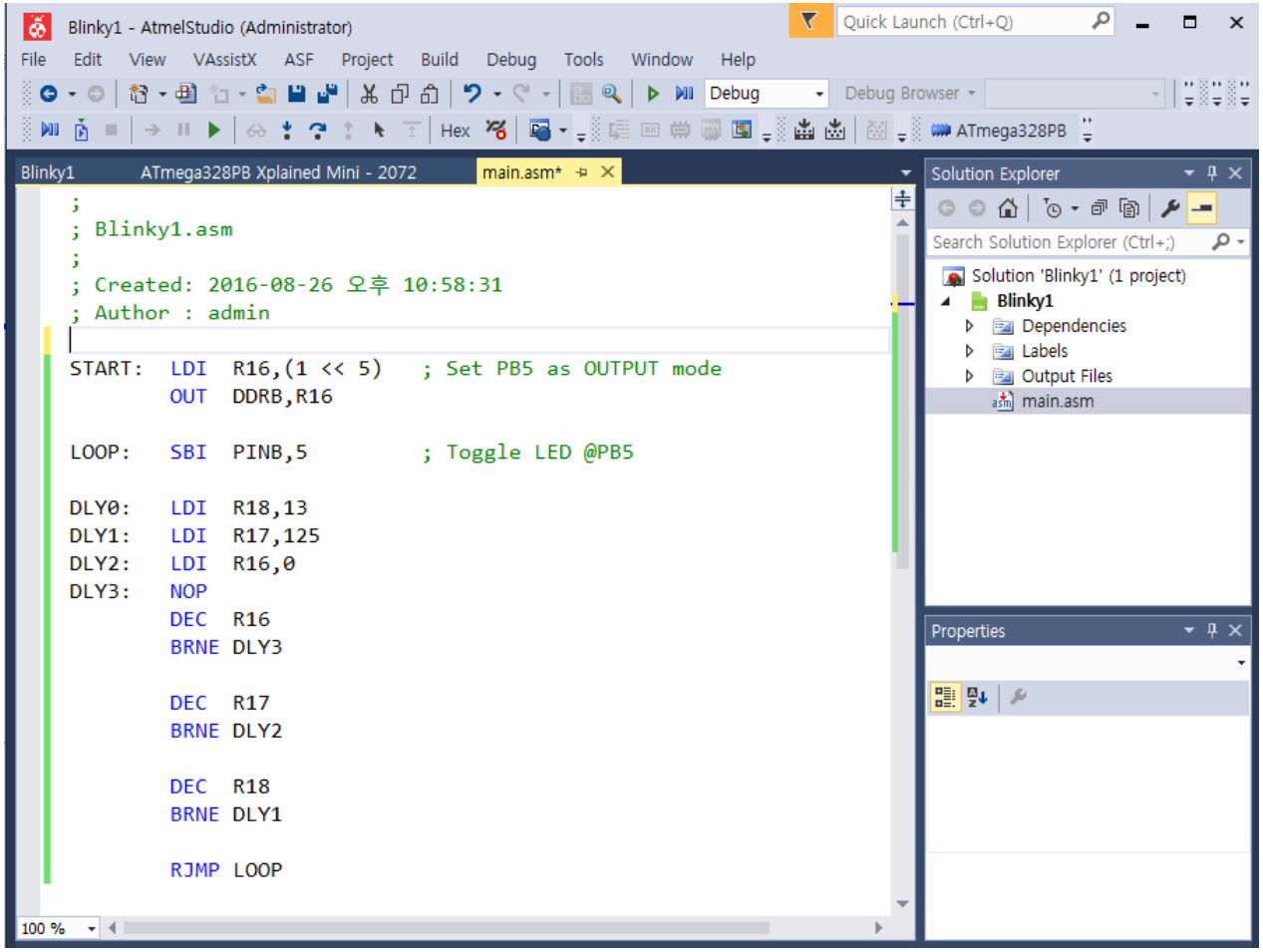
Auto-generated source program.



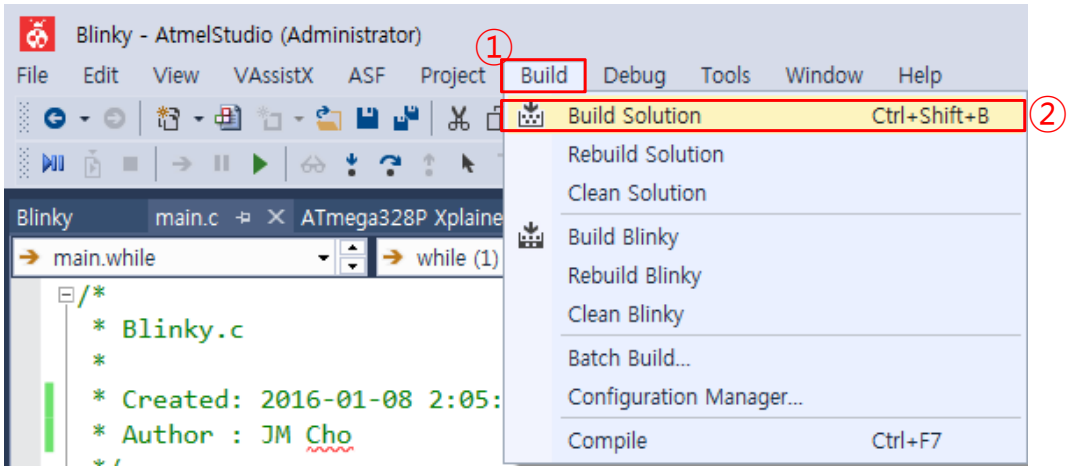
Create an AVR Application in Assembly Language (6)

Edit source file: main.asm

```
START:  LDI R16,(1 << 5)     ; Set PB5 as OUTPUT mode
        OUT DDRB,R16

LOOP:   SBI  PINB,5          ; Toggle LED @PB5

DLY0:   LDI R18,13
DLY1:   LDI R17,125
DLY2:   LDI  R16,0
DLY3:   NOP
        DEC R16
        BRNE DLY3

        DEC  R17
        BRNE DLY2

        DEC  R18
        BRNE DLY1

        RJMP LOOP
```

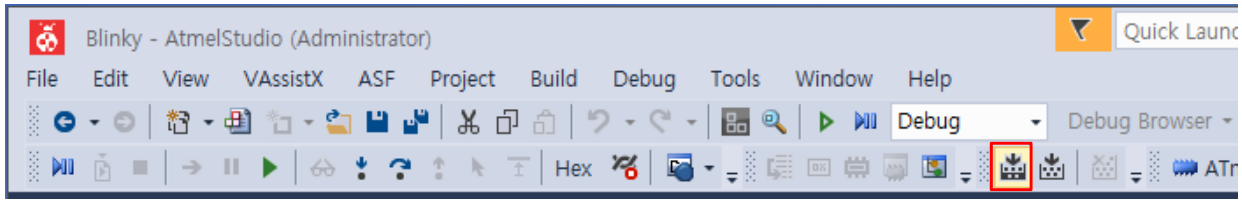


Create an AVR Application in Assembly Language (7)



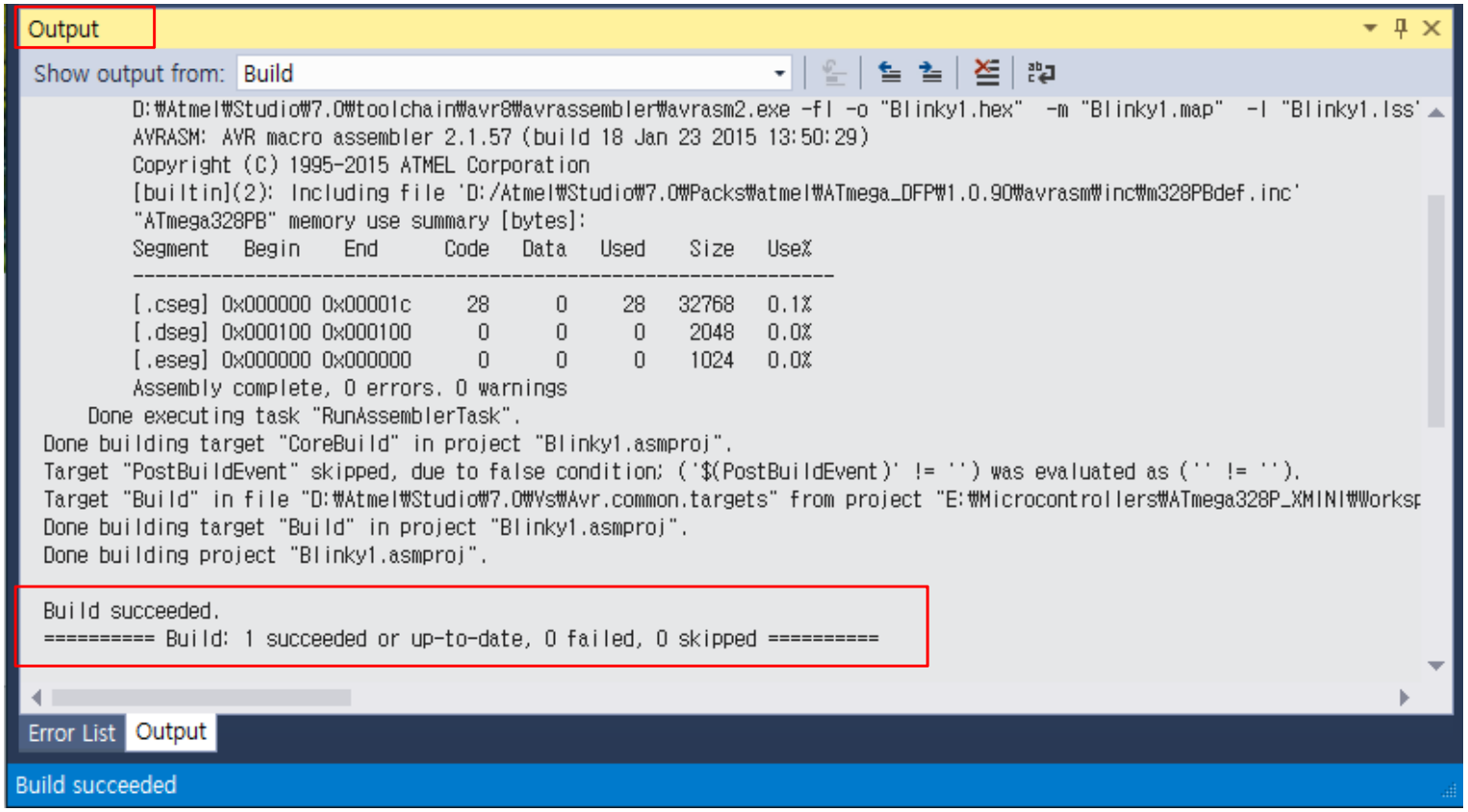
Build (Assemble and Link)

or

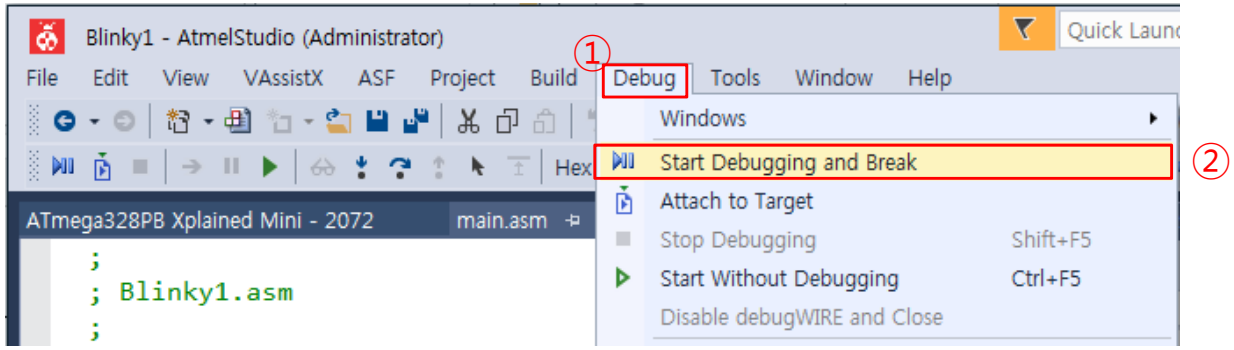


Create an AVR Application in Assembly Language (8)

Build results (succeeded case)

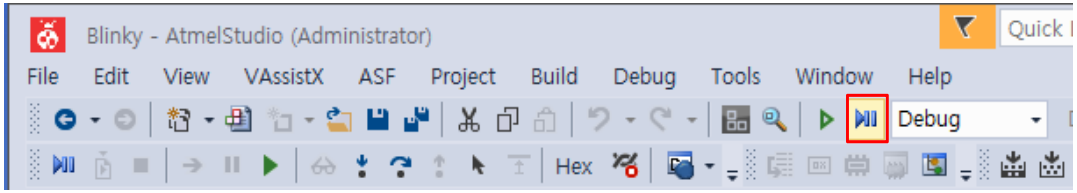


Debugging the Application (1)

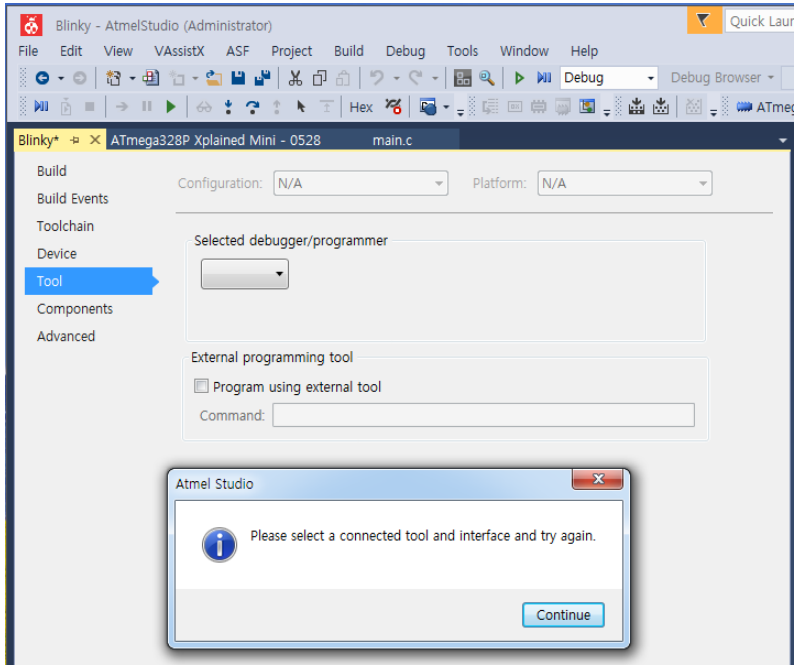


Start debugging

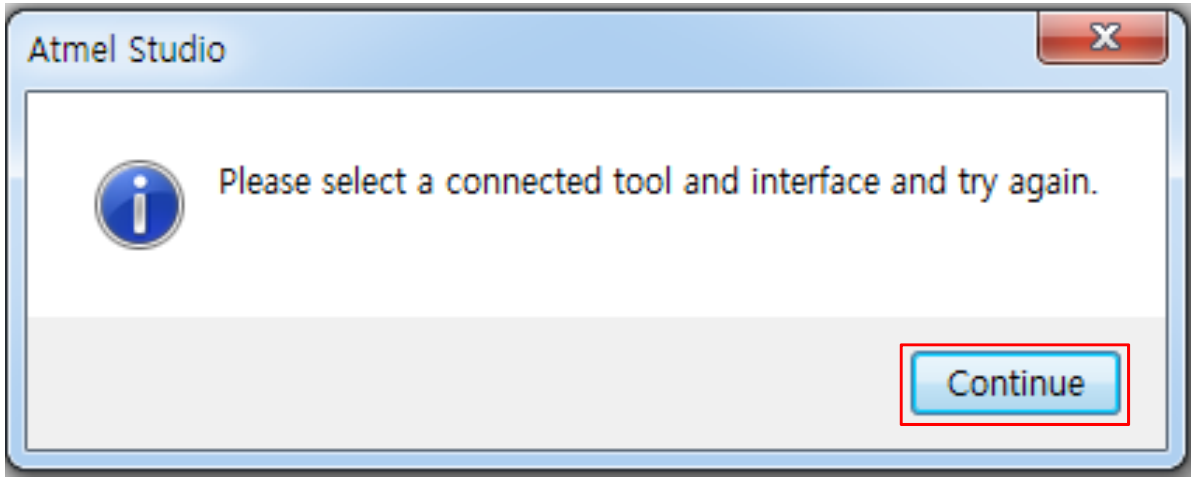
or



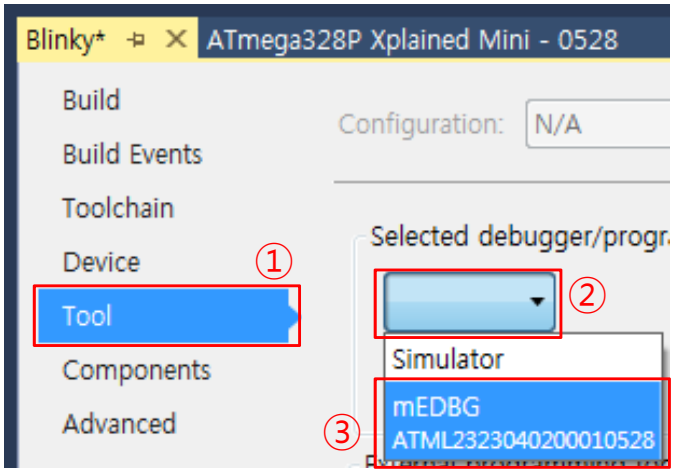
Debugging the Application (2)



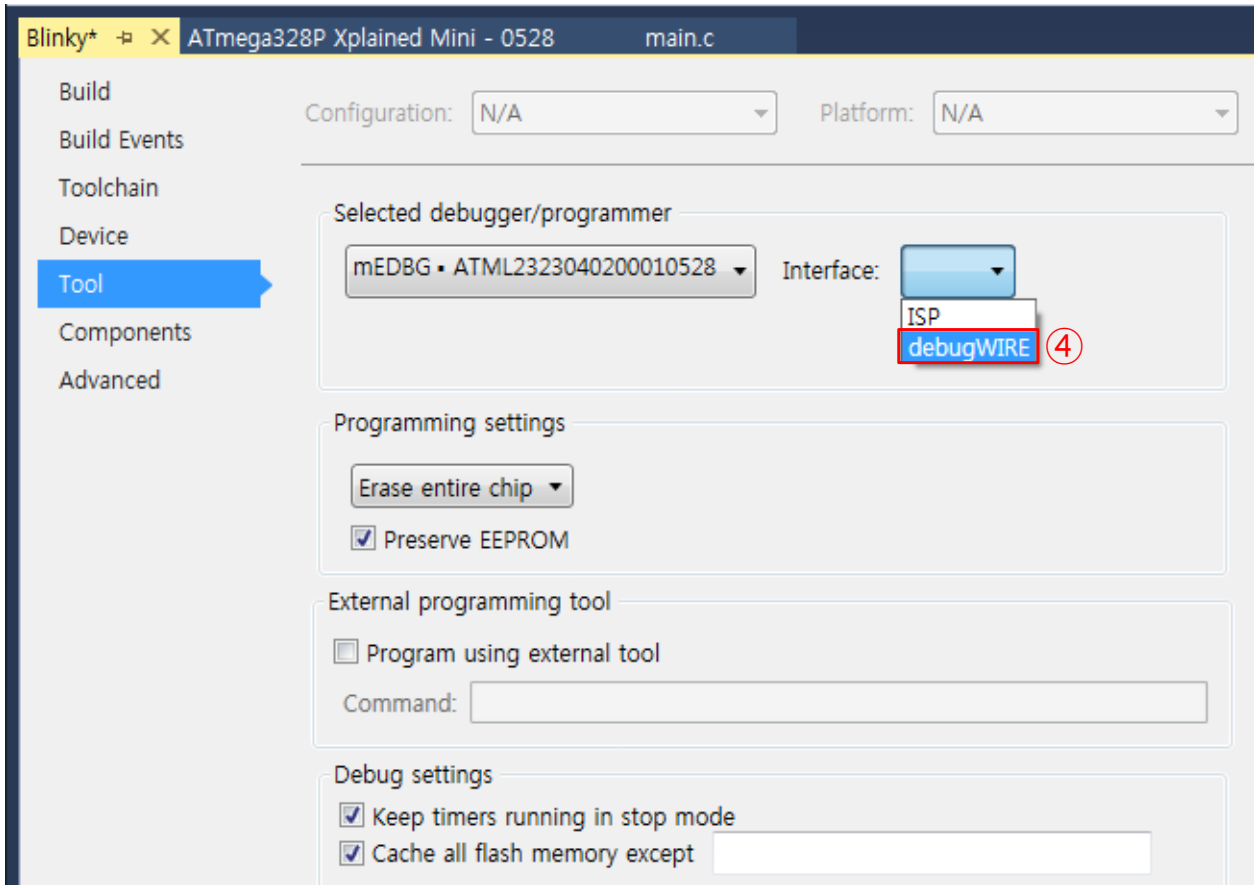
You will get this error message if you start execution of the application without selecting a debugger/programmer.



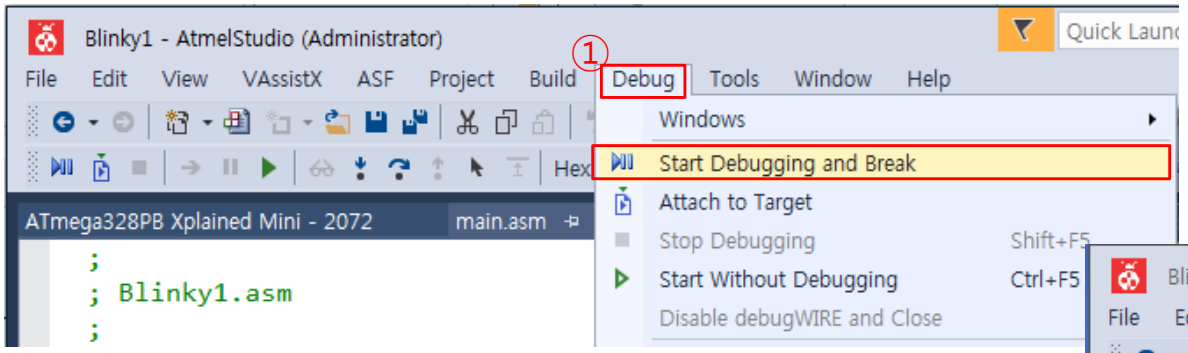
Debugging the Application (3)



Selecting a debugger/programmer

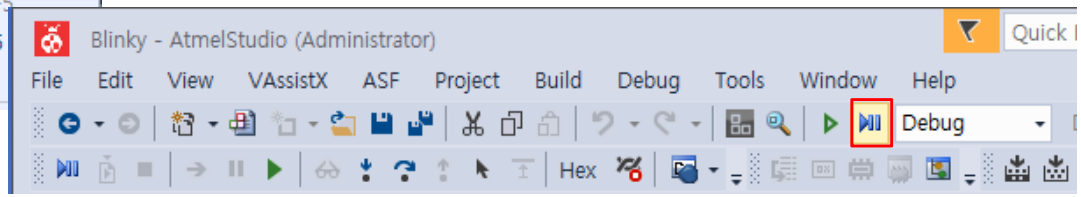


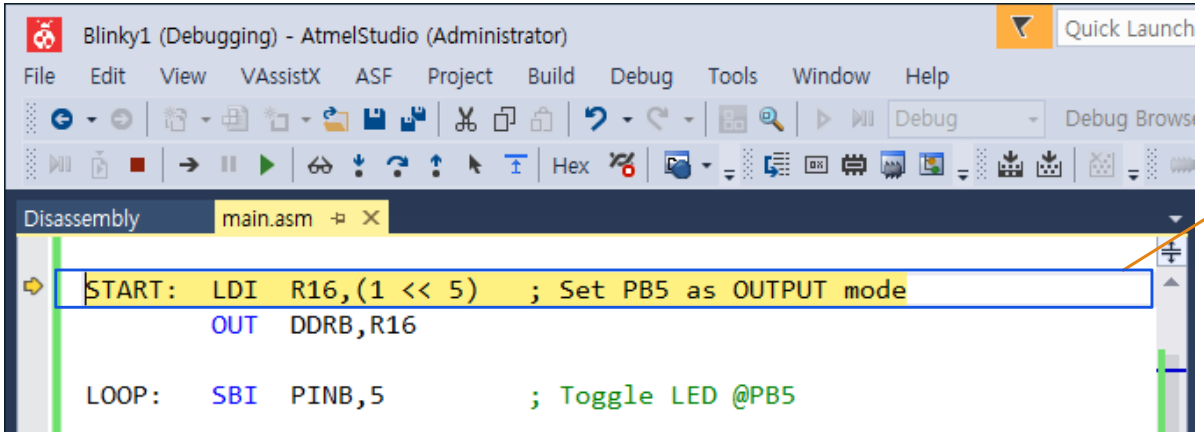
Debugging the Application (4)



Start debugging

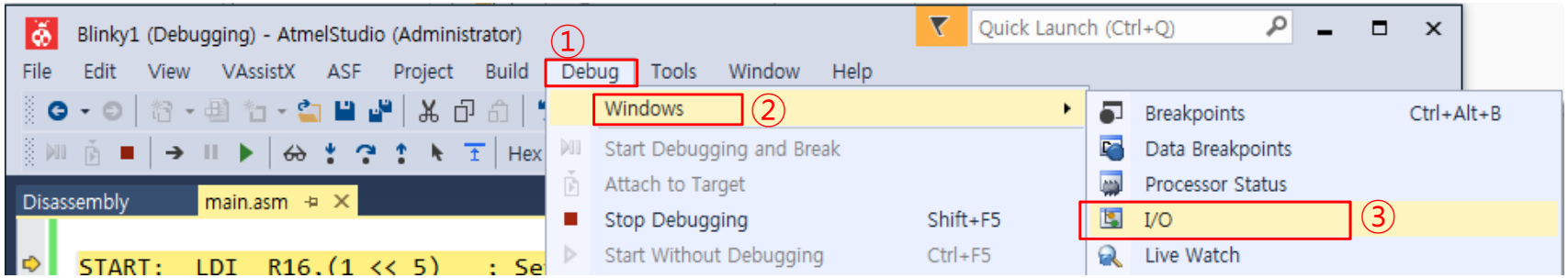
or



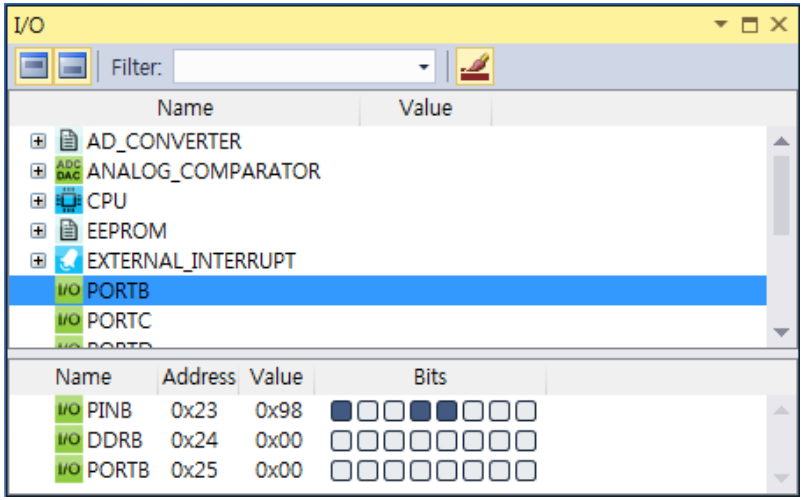


Program execution stops here and waits for user input

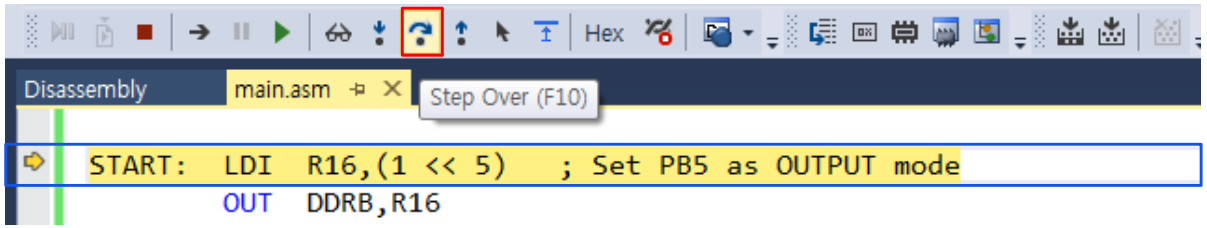
Debugging the Application (5)



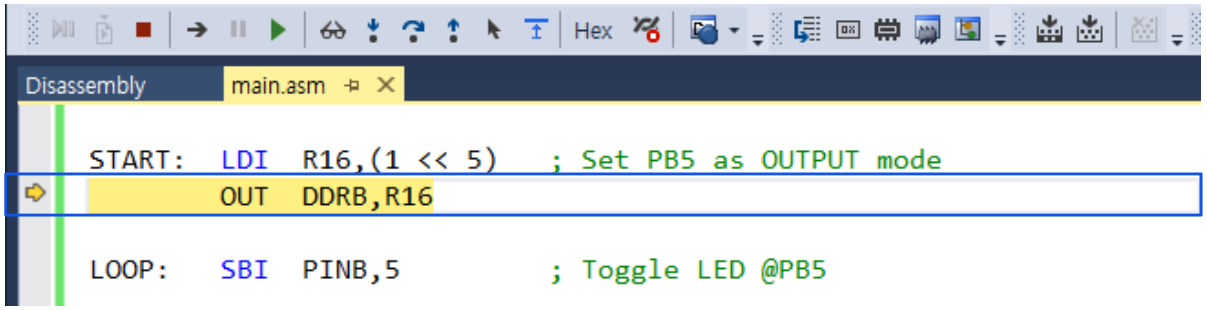
Open I/O window for debugging
(Watching registers associated to PORTB)

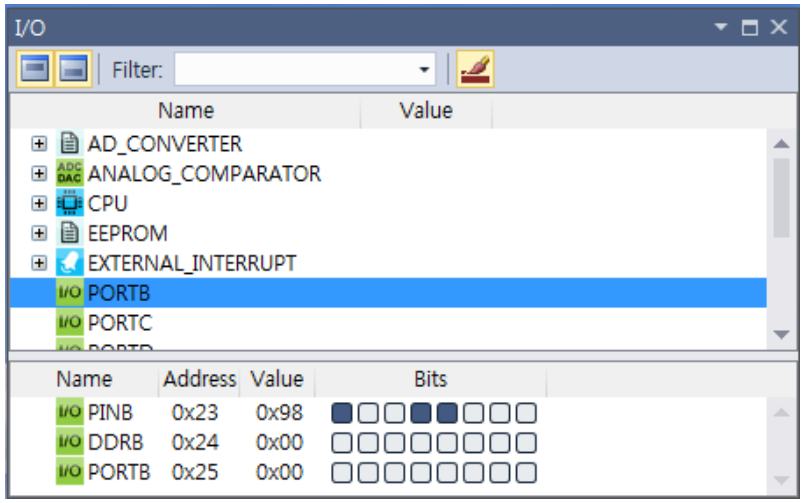


Debugging the Application (6)



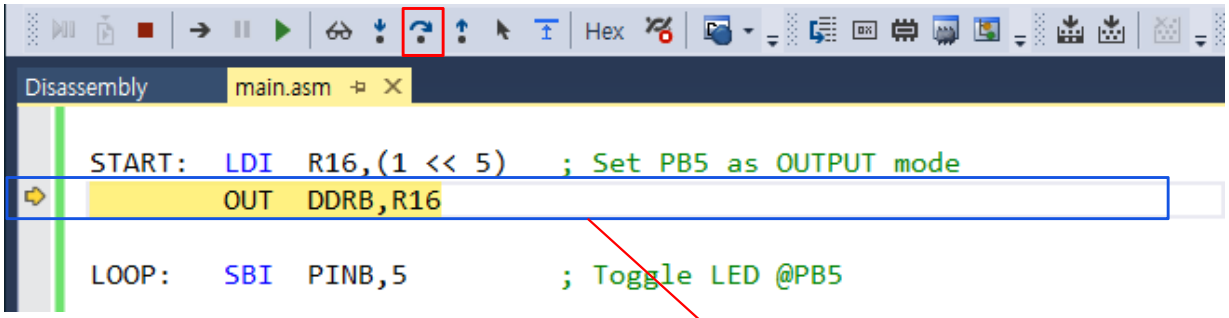
Step Over

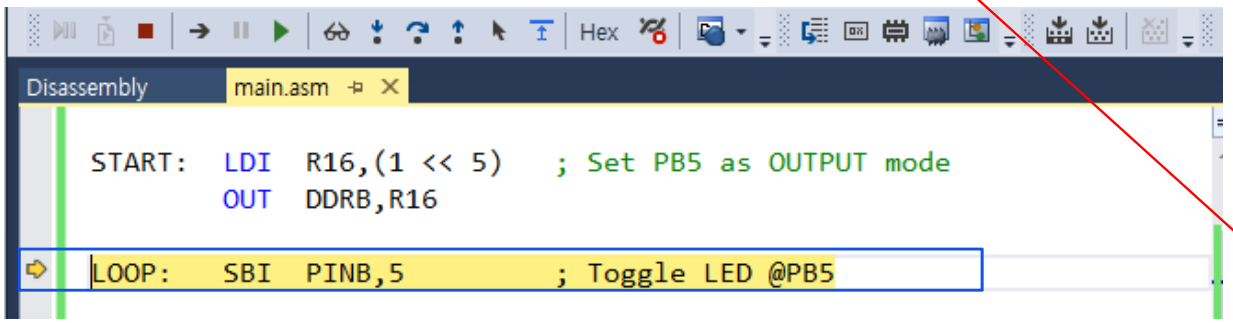


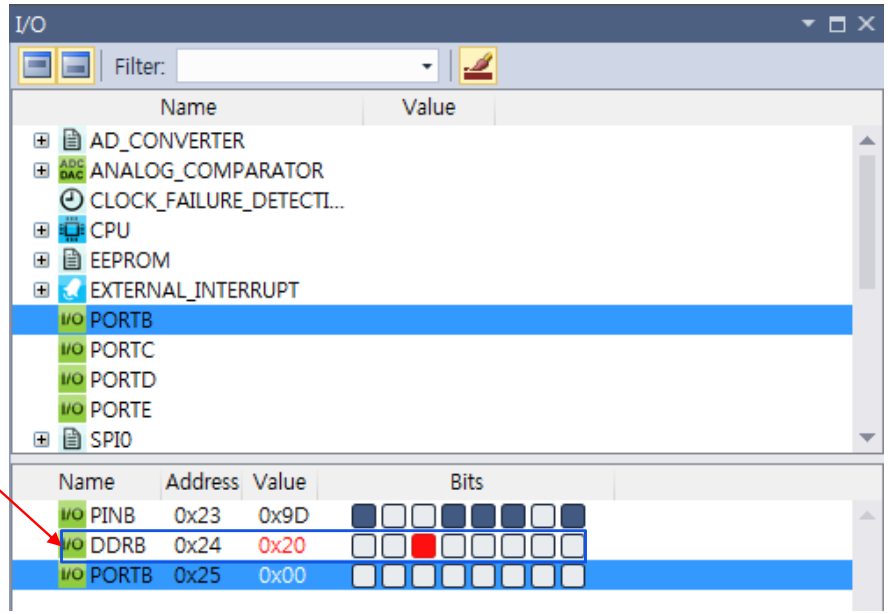


Debugging the Application (7)

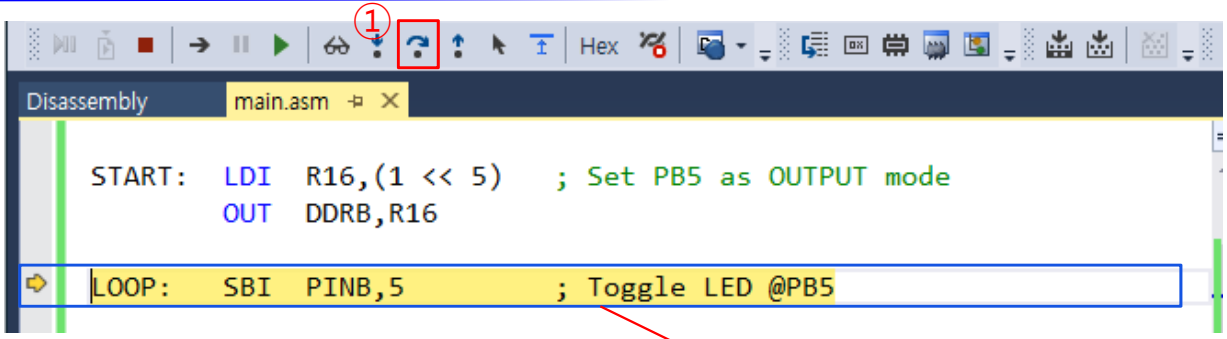
Step Over



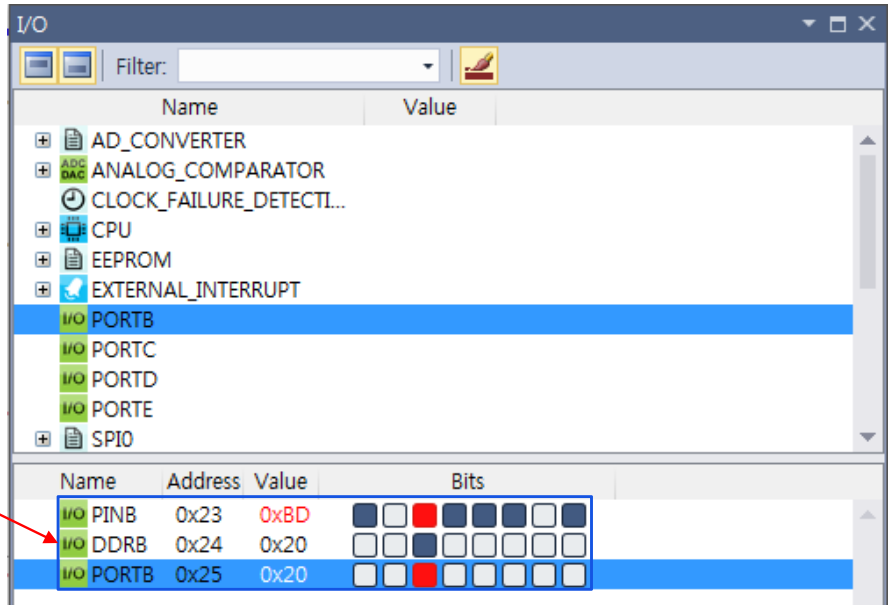


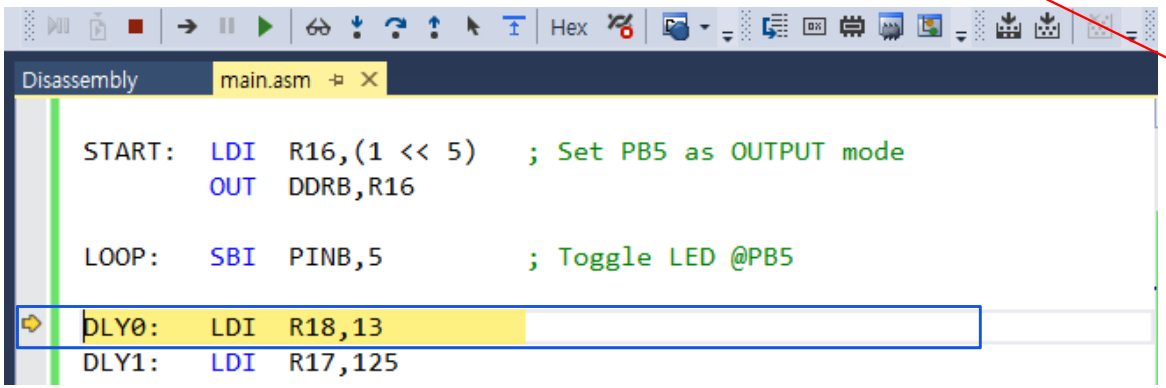


Debugging the Application (8)

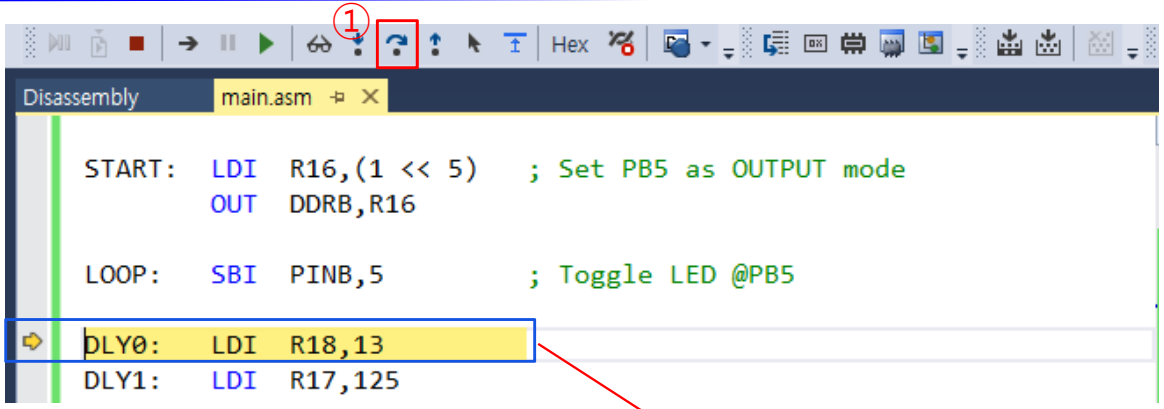


Step Over

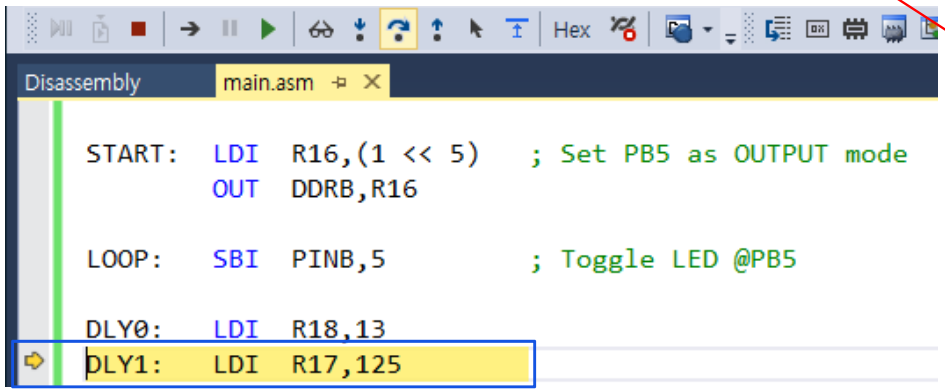




Debugging the Application (9)



Step Over

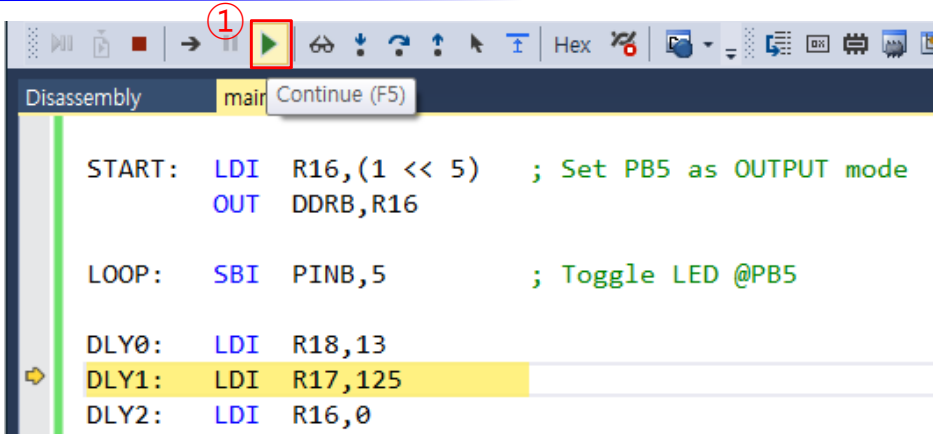


R10	0xFF
R11	0xFF
R12	0xFF
R13	0xFF
R14	0xFF
R15	0xFF
R16	0x20
R17	0x0C
R18	0x0D
R19	0x46
R20	0xFF
R21	0xFF

Processor Status

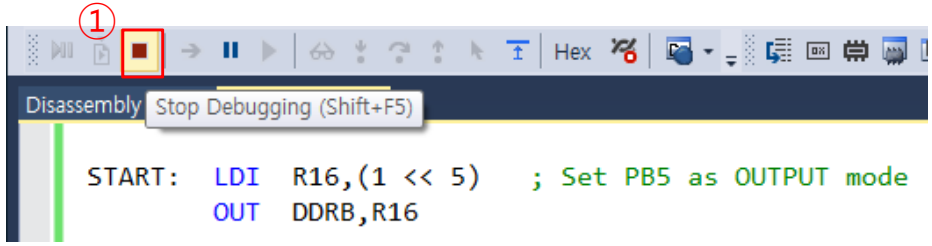
Name	Value
Z Register	0x0058
Status Register	ITHSVN
Cycle Counter	0
Frequency	
Stop Watch	
Registers	
R00	0x77
R01	0x00
R02	0xFF
R03	0xFF
R04	0xFF
R05	0xFF
R06	0xFF
R07	0xFF
R08	0xFF
R09	0xFF
R10	0xFF
R11	0xFF
R12	0xFF
R13	0xFF
R14	0xFF
R15	0xFF
R16	0x20
R17	0x0C
R18	0x09
R19	0x46
R20	0xFF

Debugging the Application (10)



Continue execution

LED blinking continues…

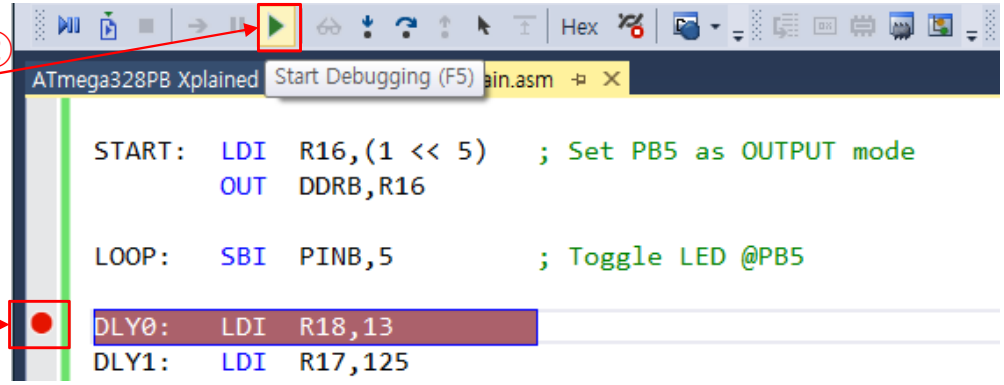


Stop debugging

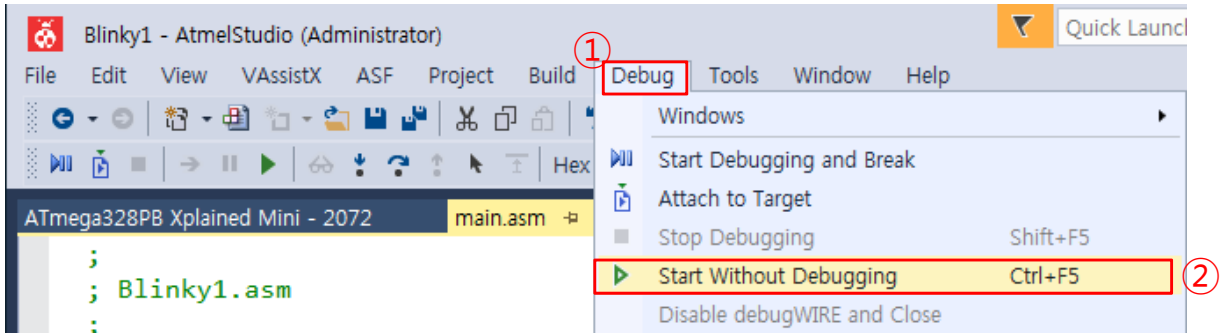
Debugging the Application (11)

Execution continues till the BREAK point

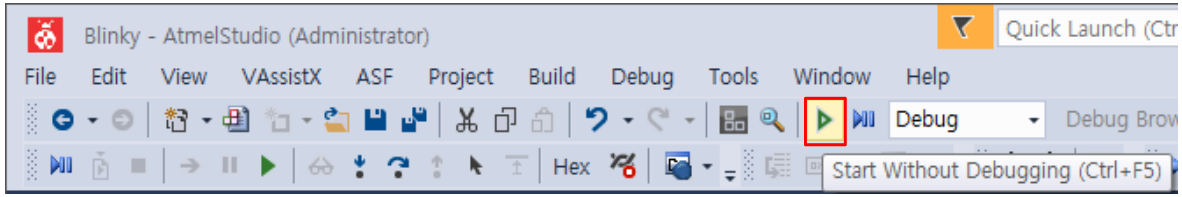
Set BREAK Point



Start the Application Without Debugging



or

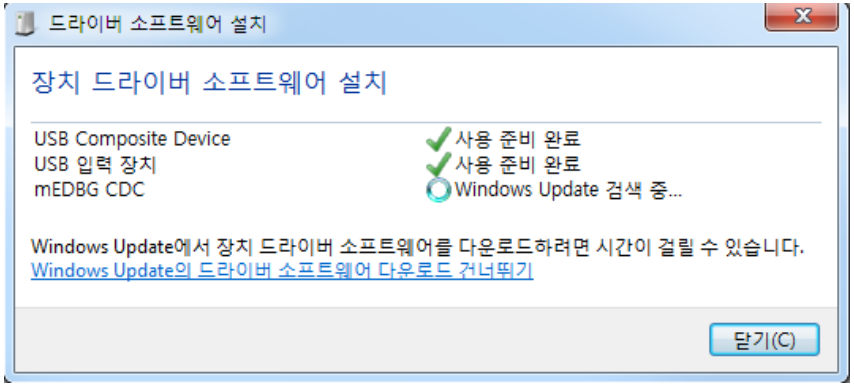


Congratulations!

You have successfully developed an AVR application in assembly language.

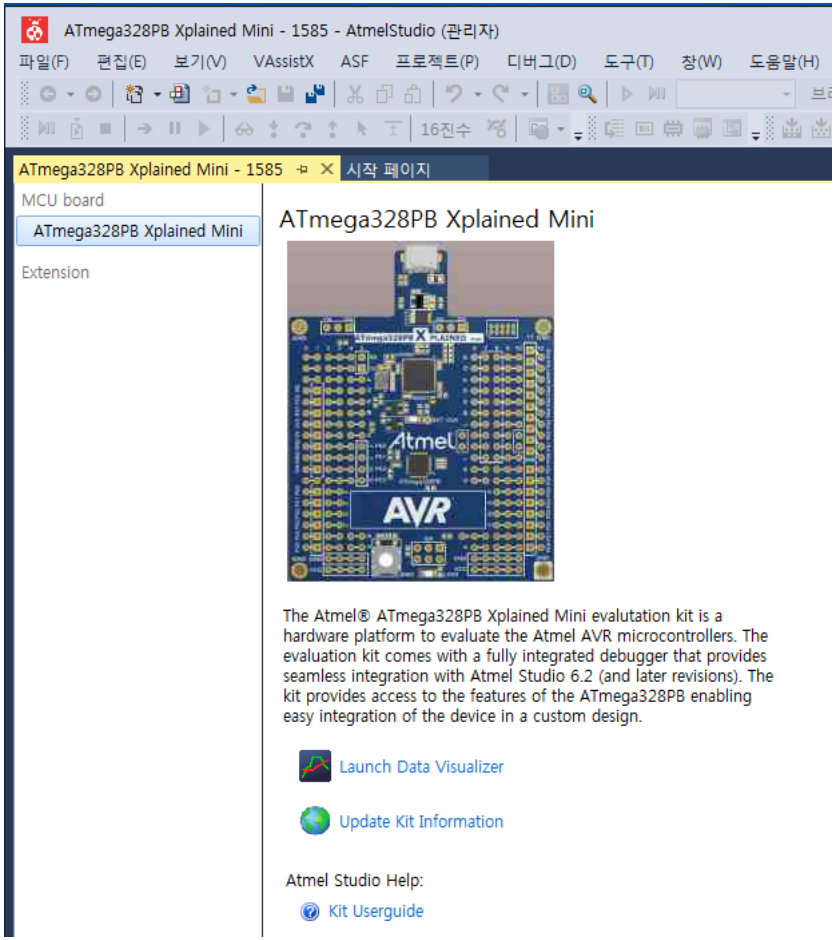
Create an AVR Application in C Language (1)

1. Connect ATmega328PB Xplained Mini to PC via USB Micro-B cable.
2. Three drivers will be installed when the board is connected for the first time.



Create an AVR Application in C Language (2)

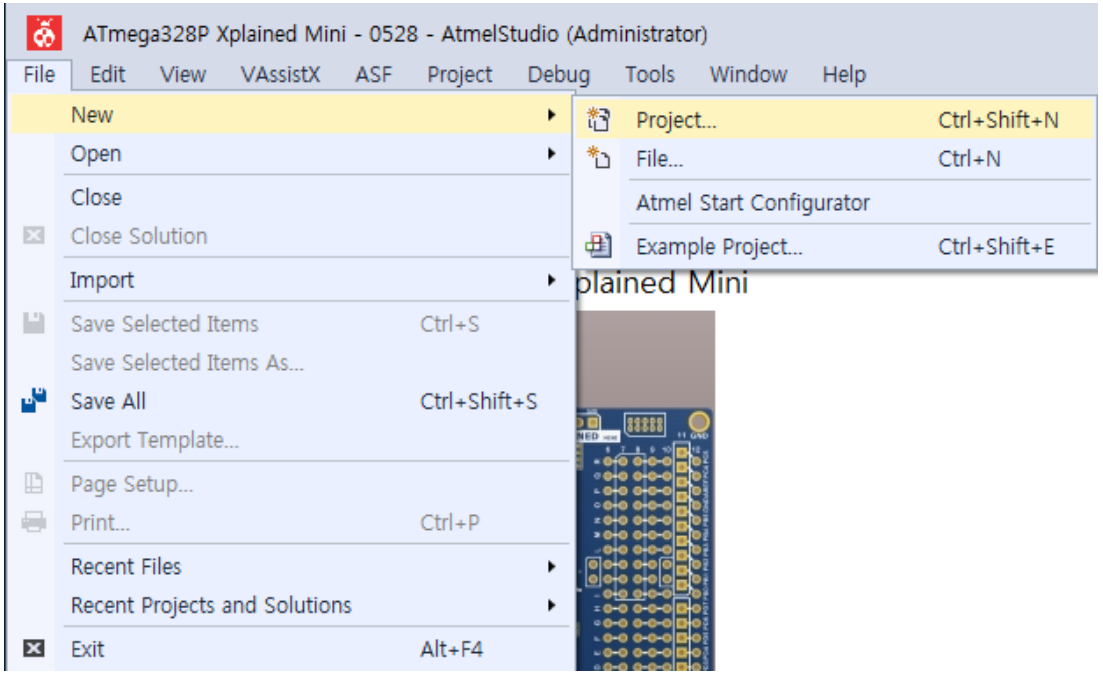
1. Run AVR Studio 7.
2. Image of the detected ATmega328PB Xplained Mini is displayed with related information.



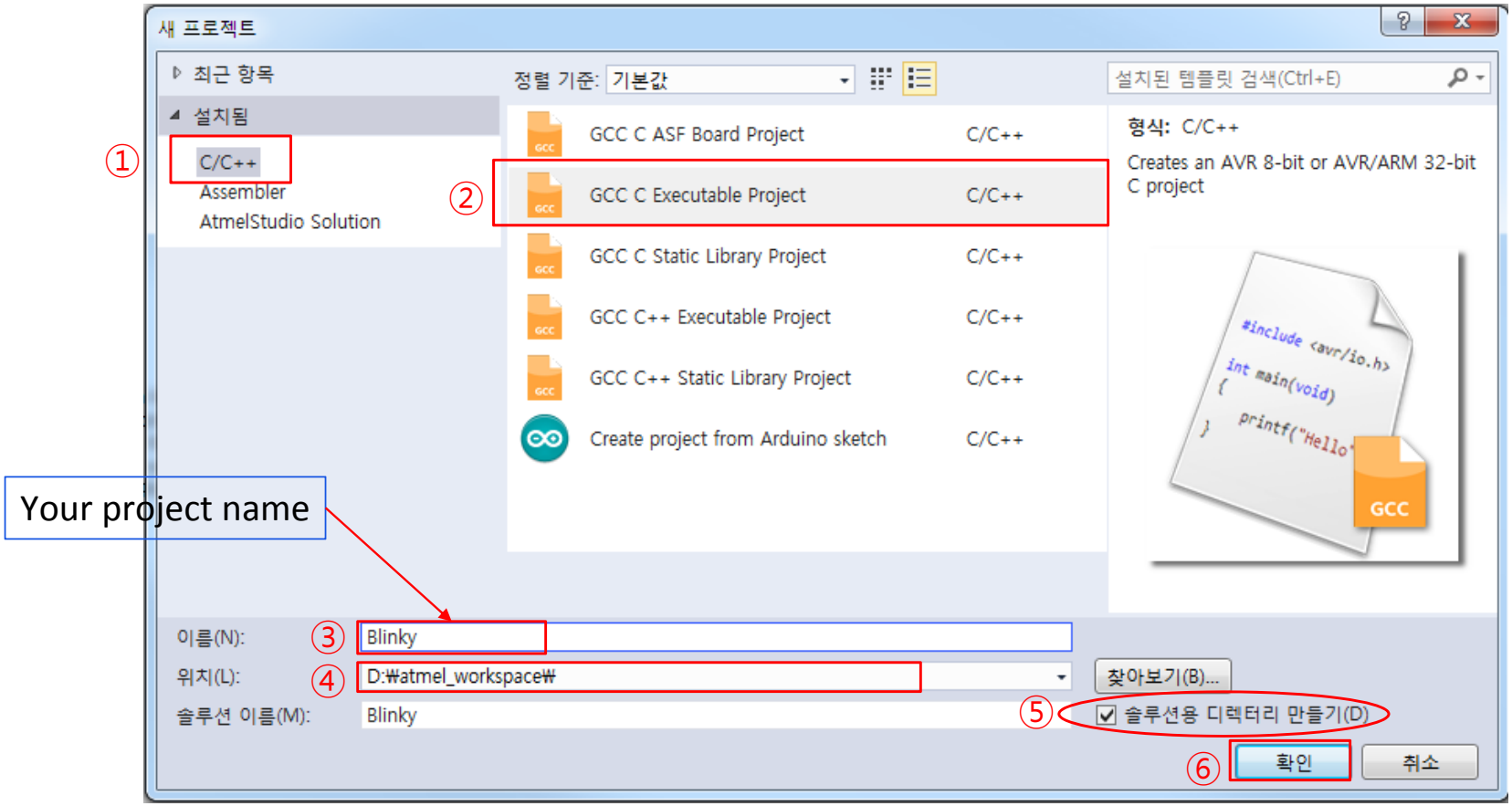
Create an AVR Application in C Language (3)

1. Select

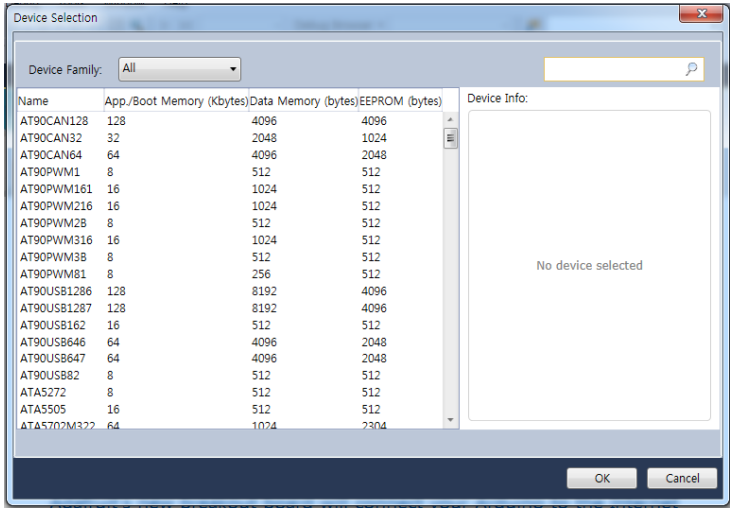
   File – New – Project…

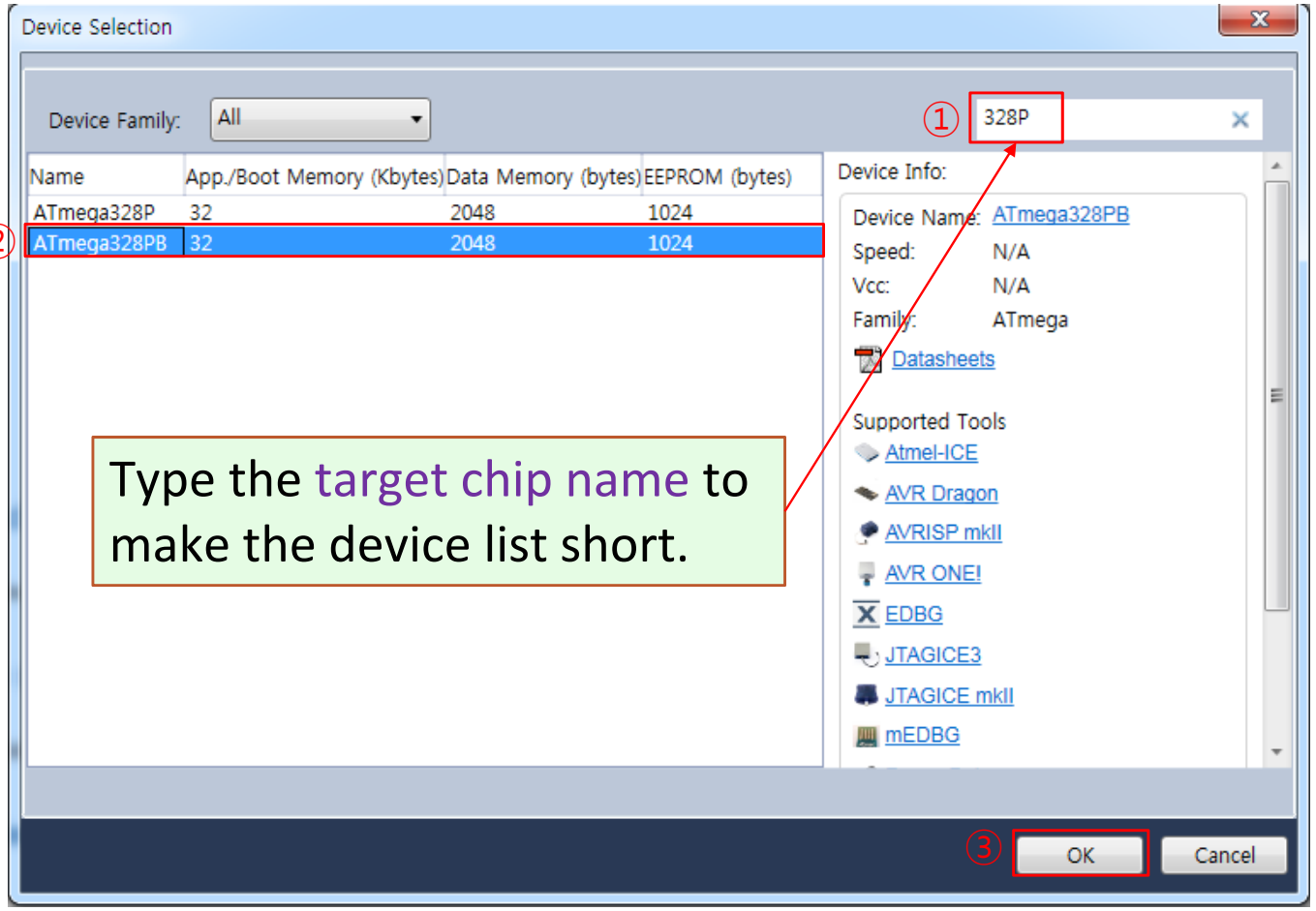


Create an AVR Application in C Language (4)



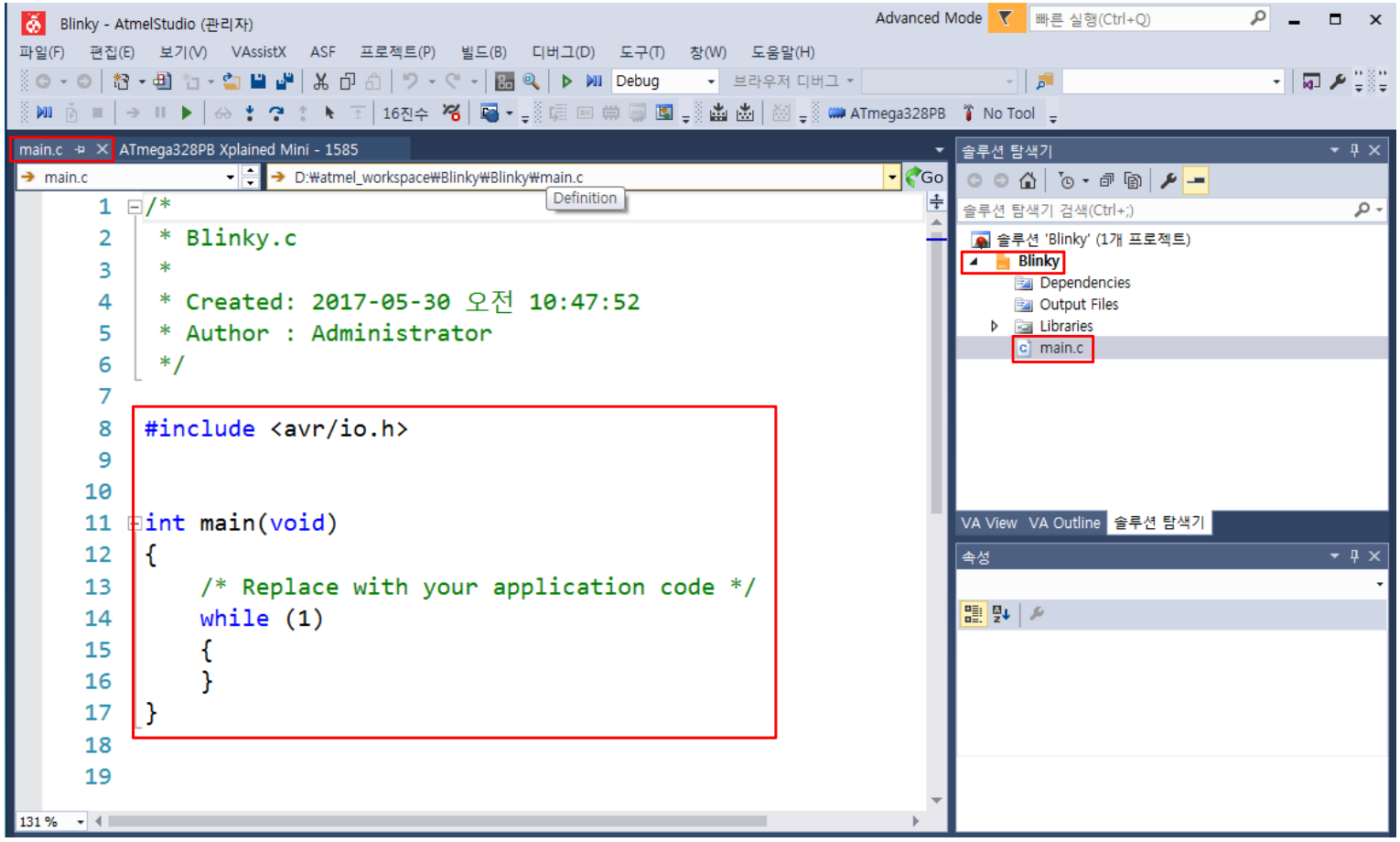
Create an AVR Application in C Language (5)





Create an AVR Application in C Language (6)

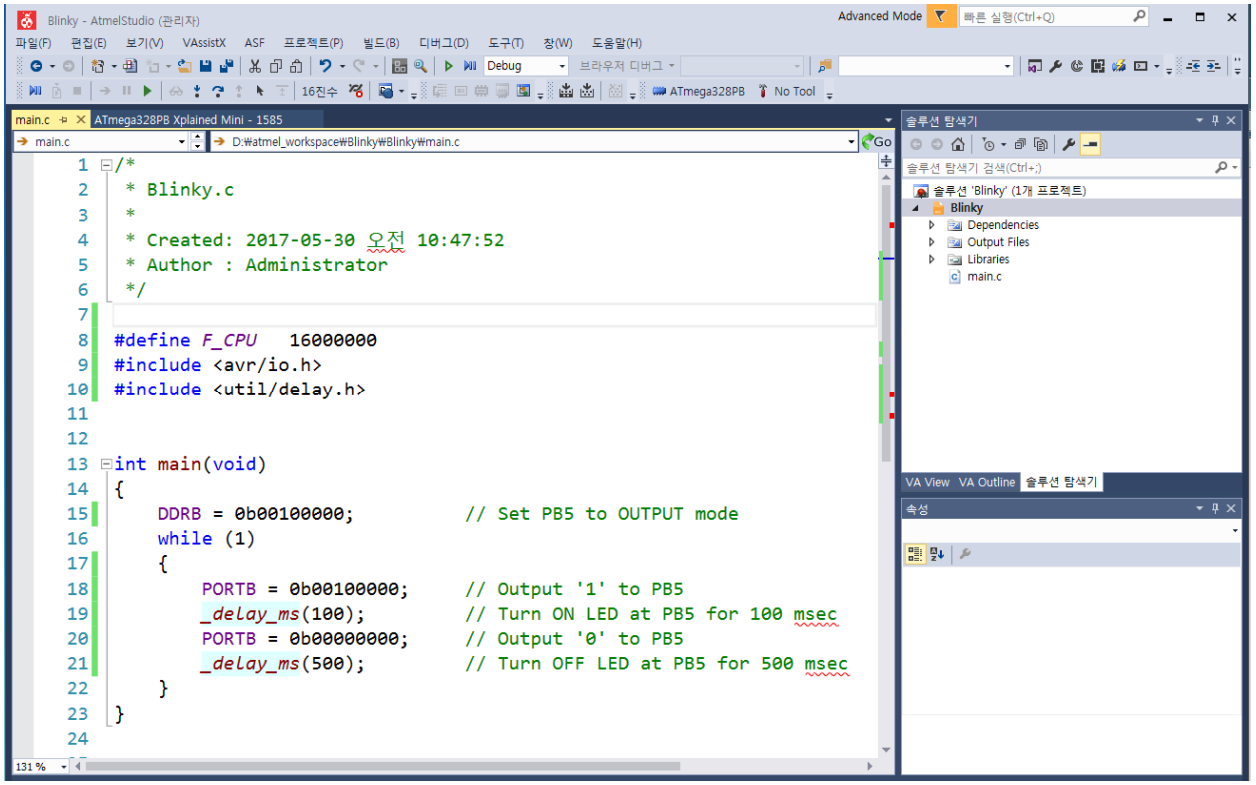
Auto-generated source program.



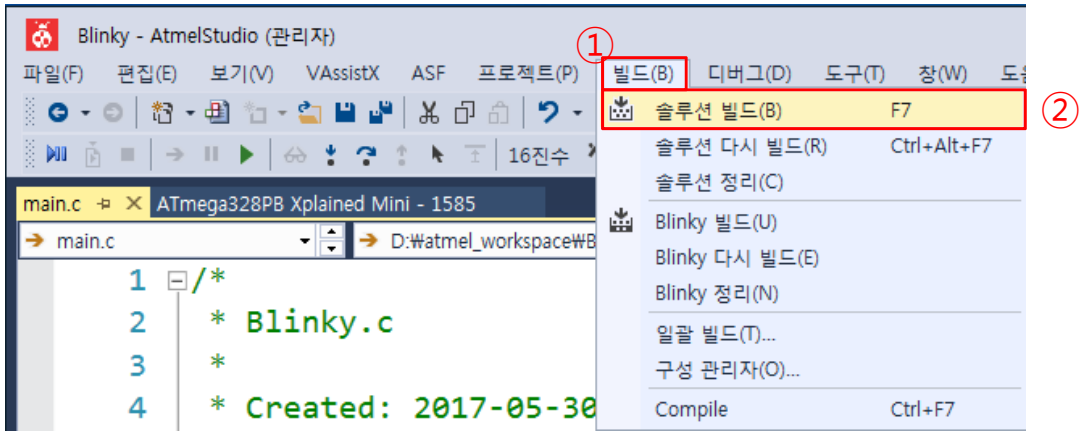
Create an AVR Application in C Language (7)

Edit source file: main.c

```
#define F_CPU16000000
#include <avr/io.h>
#include <util/delay.h>

int main(void)
{
  DDRB = 0b00100000;// Set PB5 to OUTPUT mode
  while (1)
  {
    PORTB = 0b00100000; // Output '1' to PB5
    _delay_ms(100);     // Turn ON LED at PB5 for 100 msec
    PORTB = 0b00000000; // Output '0' to PB5
    _delay_ms(500);     // Turn OFF LED at PB5 for 500 msec
  }
}
```



Create an AVR Application in C Language (8)

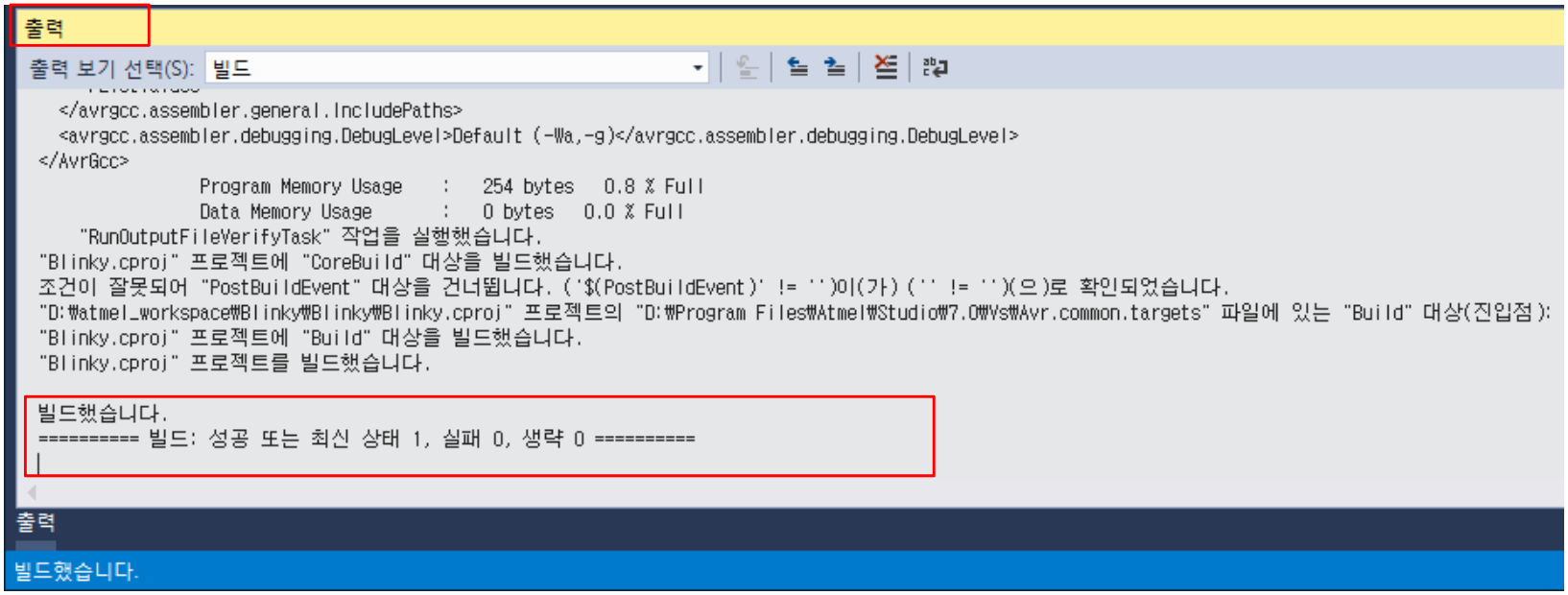


Build (Compile and Link)

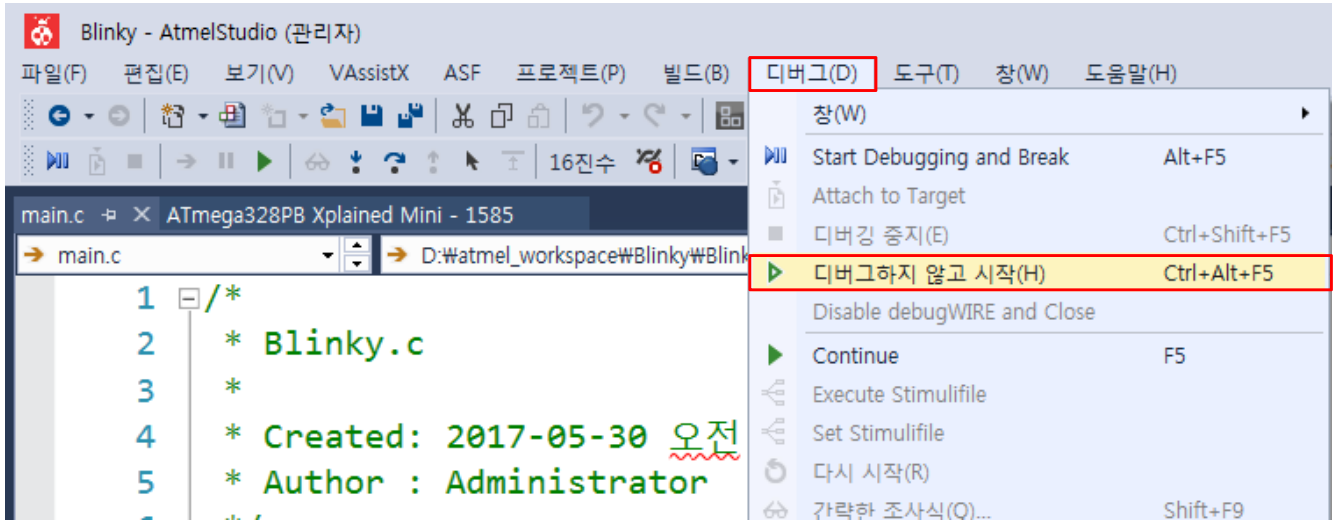
or

Create an AVR Application in C Language (9)

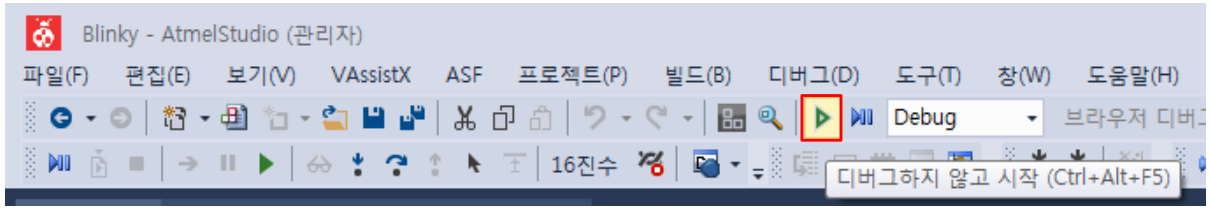
Build results (succeeded case)



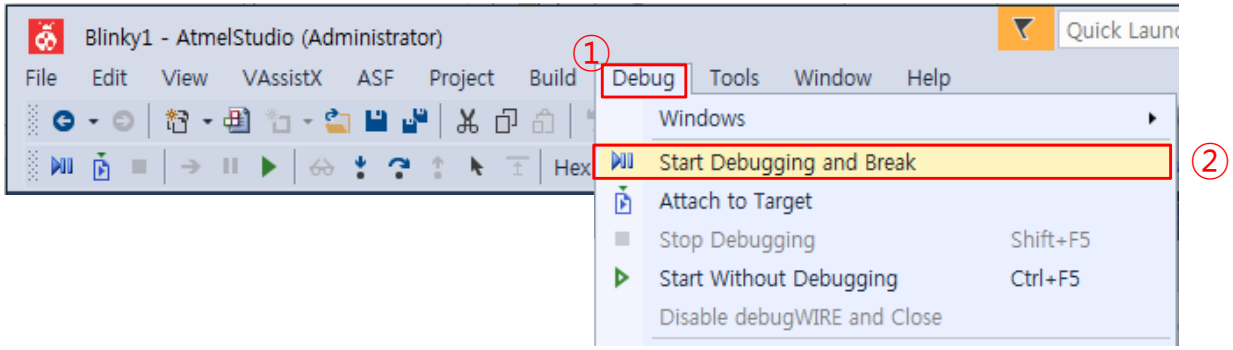
Download and Run the Application



or

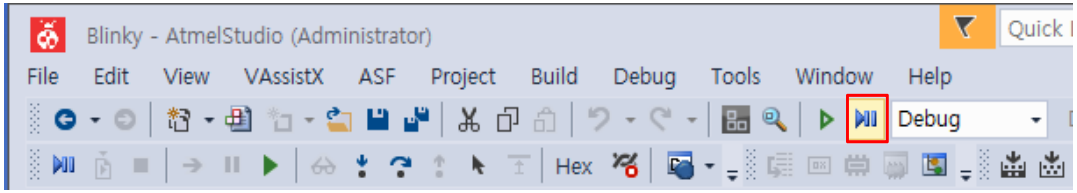


Debugging the Application (1)

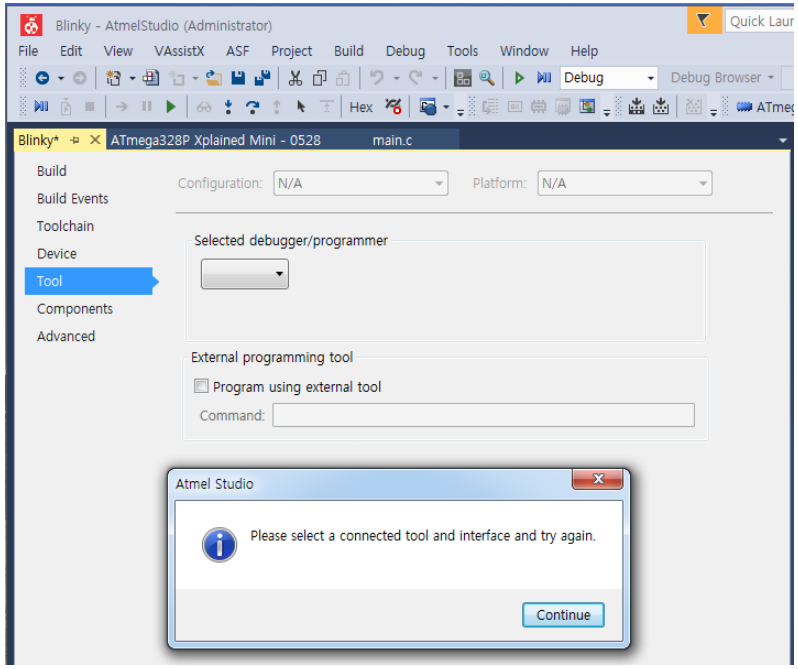


Start debugging

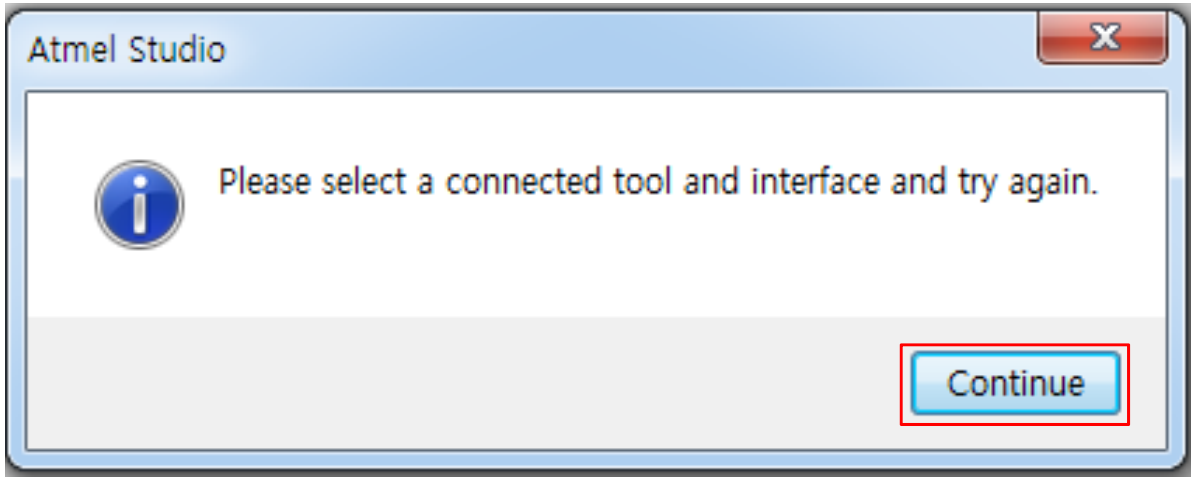
or



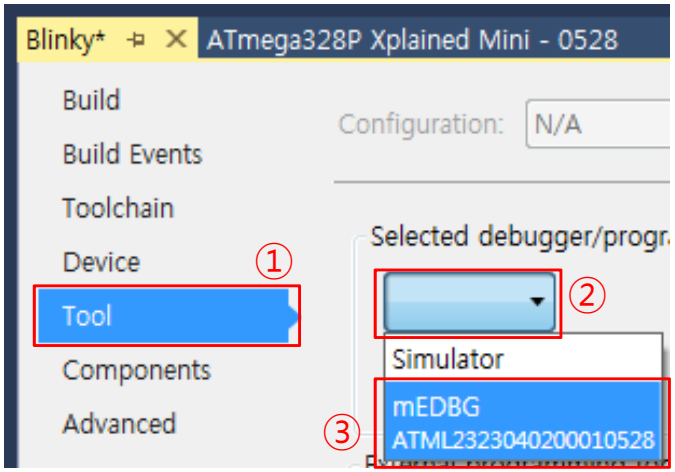
Debugging the Application (2)



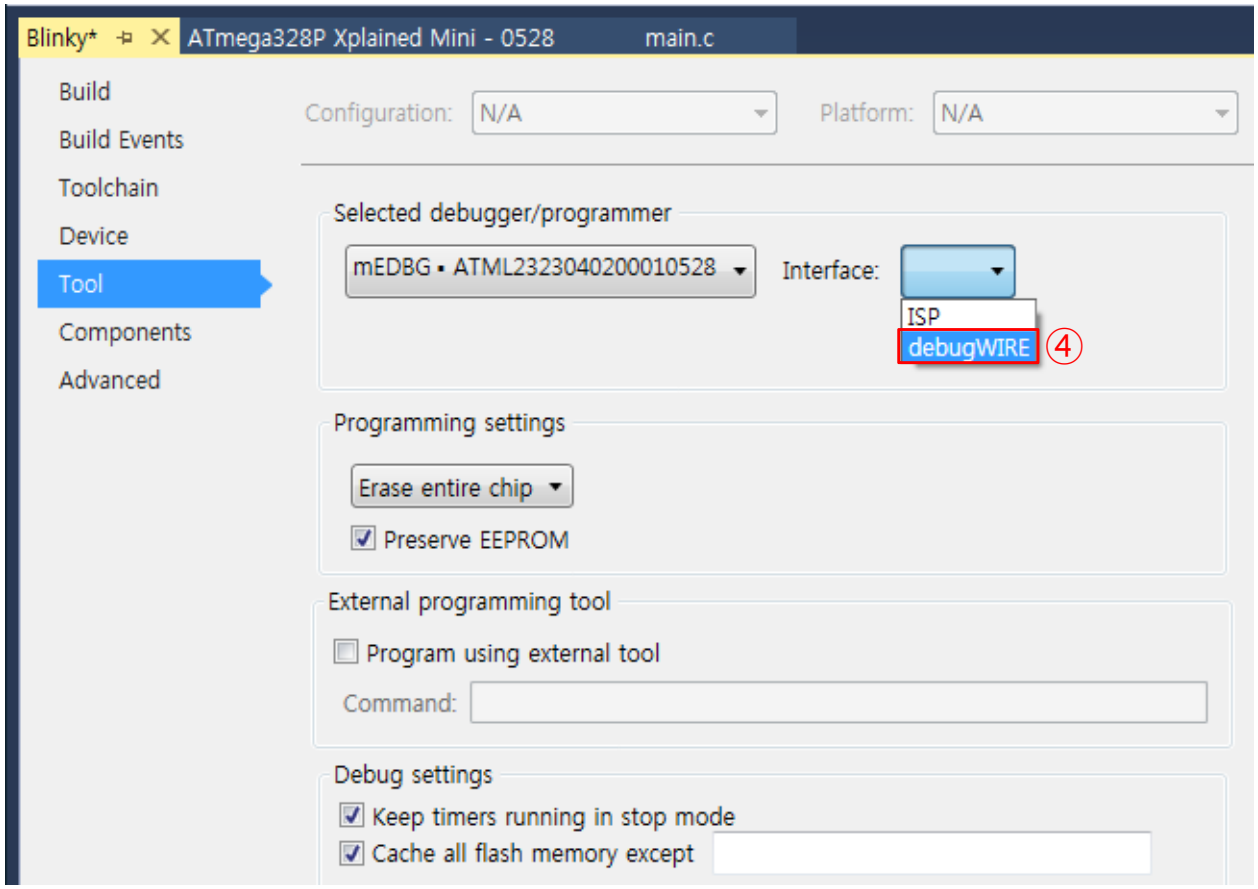
You will get this error message if you start execution of the application without selecting a debugger/programmer.



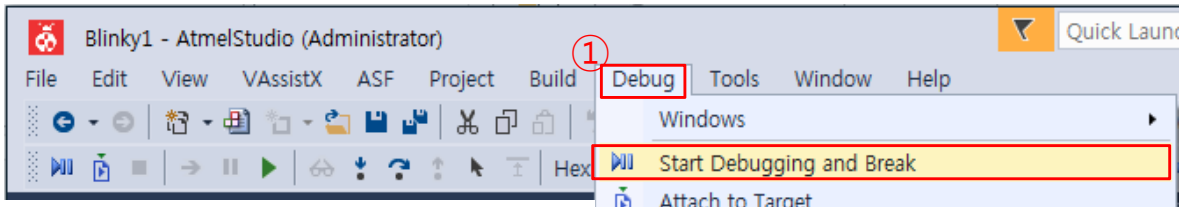
Debugging the Application (3)



Selecting a debugger/programmer



Debugging the Application (4)

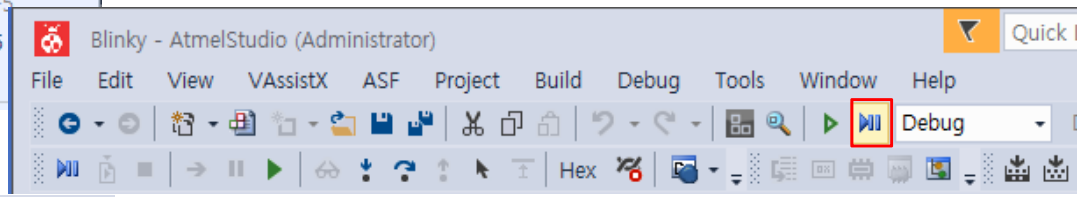


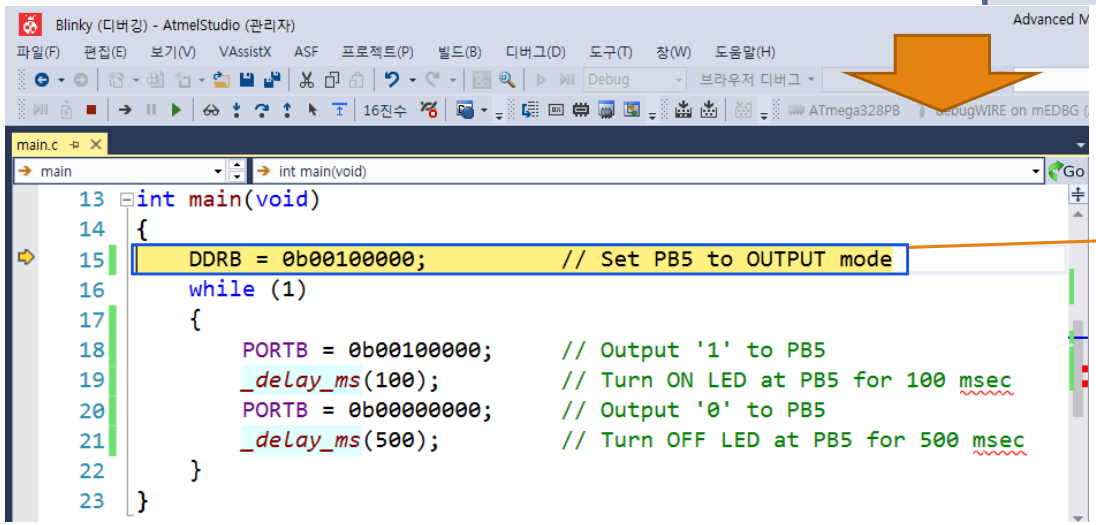
①

②

or

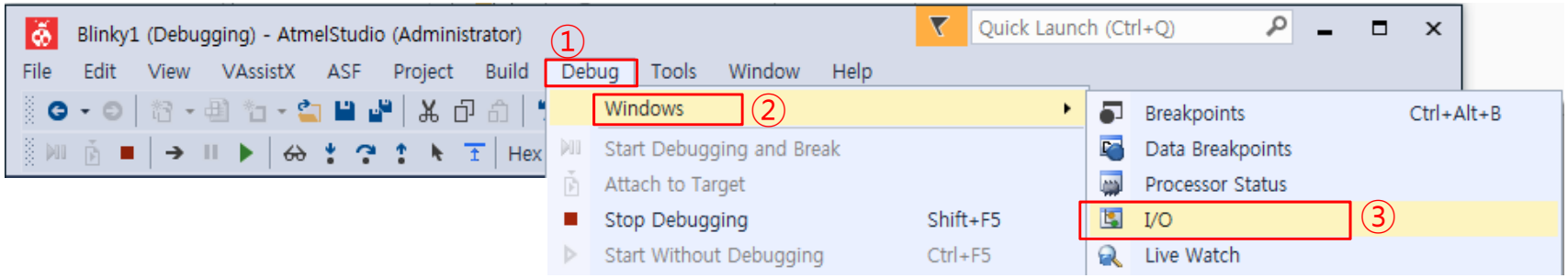
Start debugging



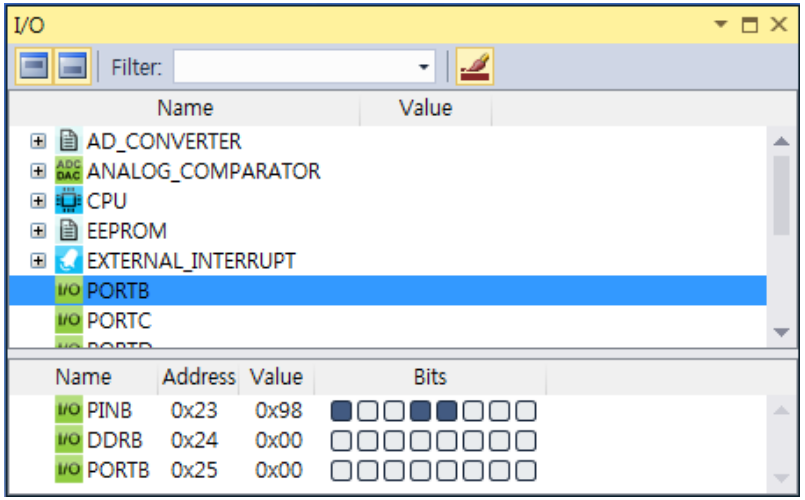


Program execution stops here and waits for user input

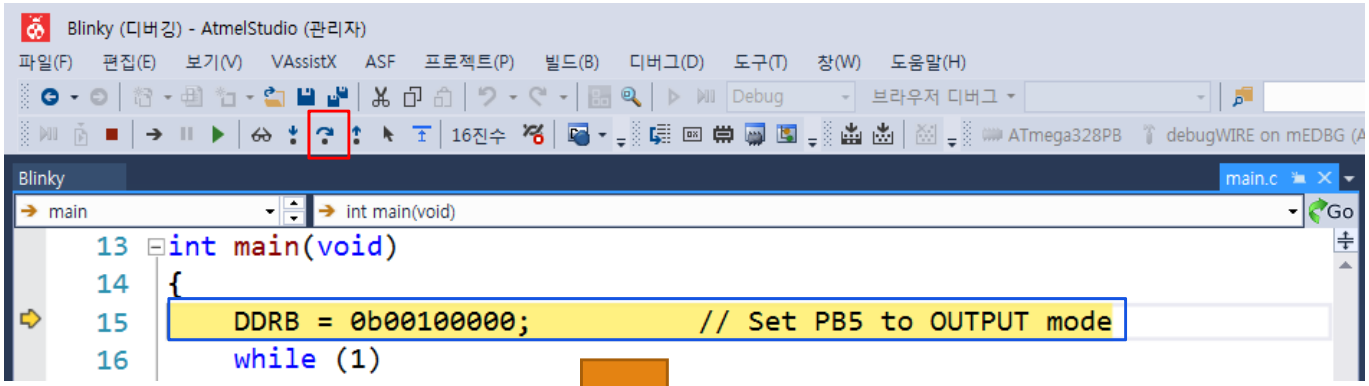
Debugging the Application (5)



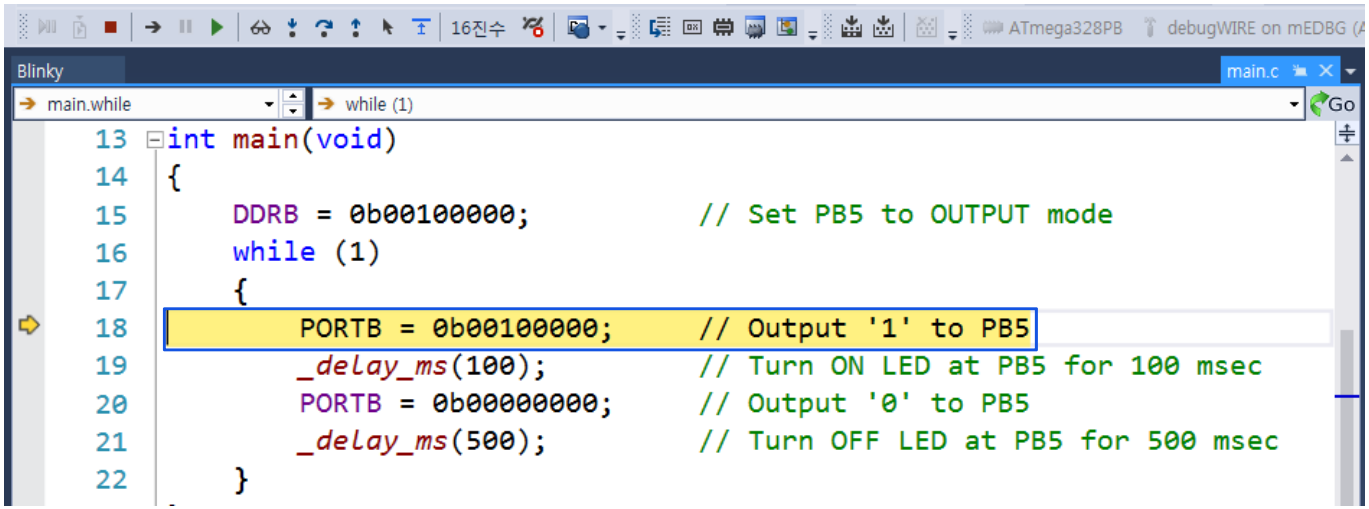
Open I/O window for debugging
(Watching registers associated to PORTB)

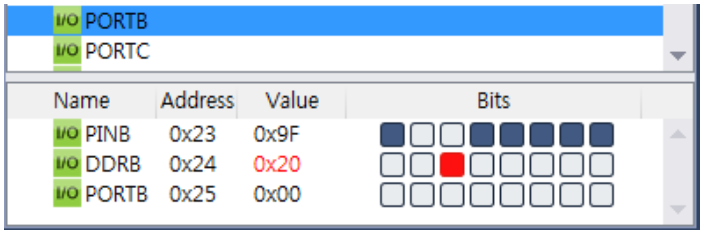


Debugging the Application (6)

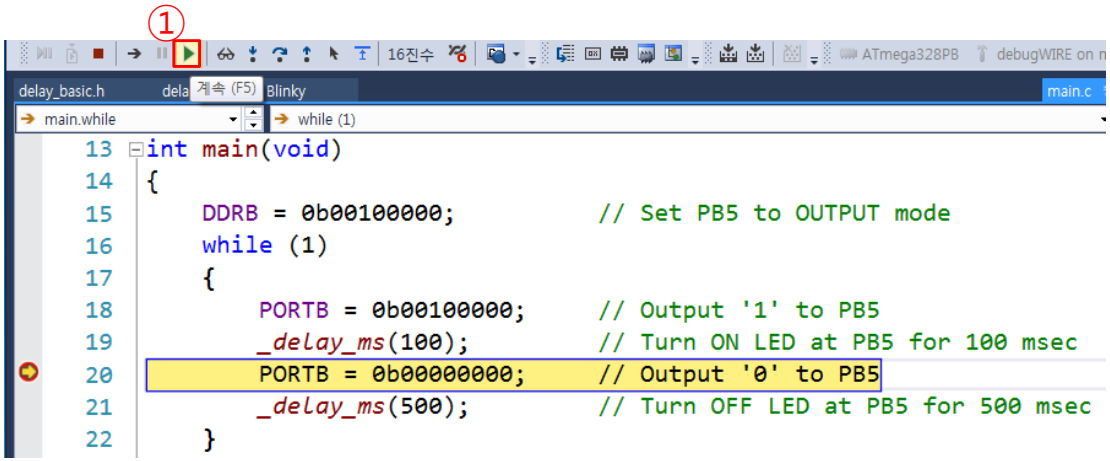


Step Over



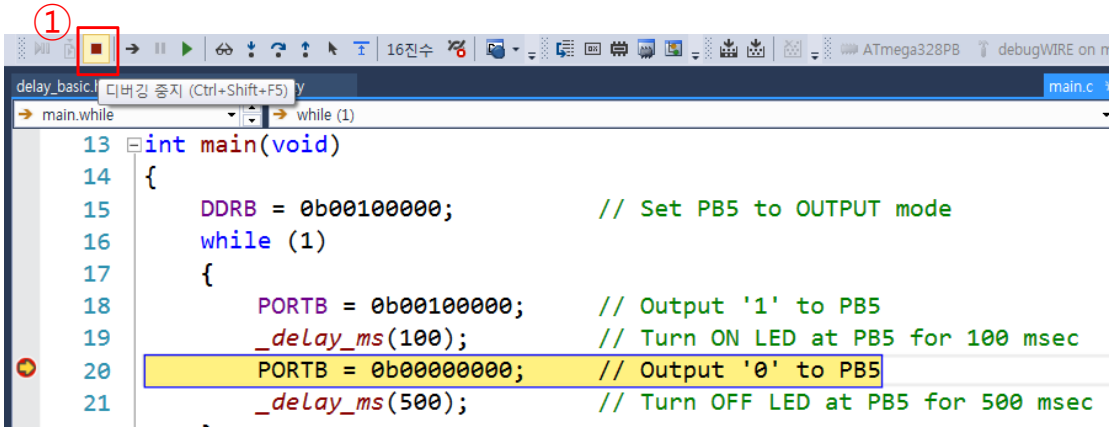


Debugging the Application (10)



Continue execution

LED blinking continues…



Stop debugging

Congratulations!

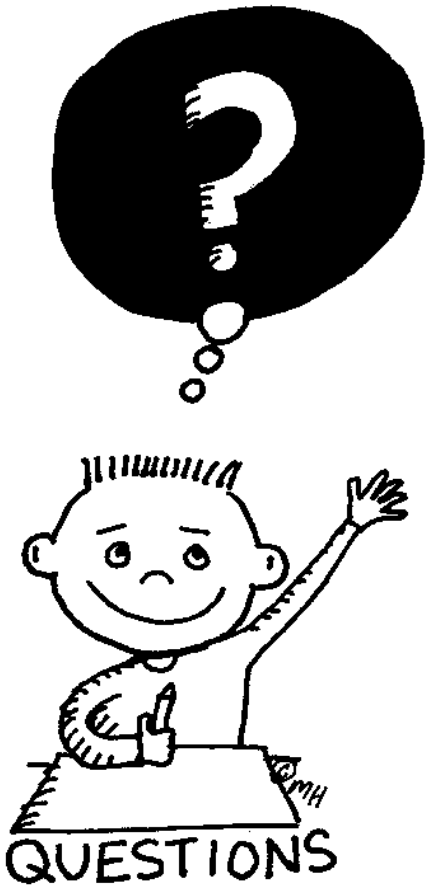
You have successfully developed an AVR application in C language.

What's next?


THE TWO STATES OF EVERY PROGRAMMER
I AM A GOD.
I HAVE NO IDEA WHAT I'M DOING.

Summary

- ATmega328PB instruction set
- Assembly language directives
- Assembly language expressions
- Mixed Language Programming


QUESTIONS


WHAT'S
NEXT?