

Arminno™

內容

前言	12
歡迎	12
微控制器無所不在	12
利基的平台	12
系統概述	14
簡介	14
C++ 程式語言	14
Arminno™ 單板電腦	15
Arminno™ 內建硬體功能及相關函式庫支援	15
CMDiBus™ 週邊模組及函式庫	15
安裝與開始使用	17
簡介	17
安裝 Keil µVision IDE 開發環境	17
安裝 e-Link32 USB Driver	24
安裝 e-Link32 Keil Plugin	31
如何編寫第一個程式	33
開啟(Open)專案	33
建置(Build)專案	34
開啟偵錯模式	35
執行程式	35
剛剛完成了些什麼?	36
Keil µVision IDE 整合開發環境	37
簡介	37
視窗各部解說	37
一般模式	37
除錯模式	37
程式開發步驟	38
選單及命令	43
硬體說明	44
簡介	44
Arminno™ 單板電腦	44
Arminno™ 系統主要功能介紹 :	44
Arminno™ 應用腳位說明	47
Arminno Workshop 板應用腳位說明	48
CMDiBus™ 週邊模組	50
靜電預防措施	51

C++ 程式語言	52
簡介	52
Hello World 程式	52
架構及敘述 (Statements)	52
註釋 (Comments)	53
識別字 (Identifiers)	53
關鍵字 (Keywords)	54
標籤 (Labels)	54
常數、變數及資料型態 (Constants, Variables and Data Types)	54
型態轉換	55
整數值 (Integral Literals)	56
浮點數值 (Floating-Point Literals)	56
陣列 (Arrays)	56
指標	57
運算子 (Operators)	57
算術運算子	58
關係運算子	58
位元運算子	58
邏輯運算子	59
指定運算子	59
程式控制流程	59
if 敘述	60
switch...case 敘述	60
while 敘述	61
do...while 敘述	61
for 敘述	61
goto、continue、break 及 return 敘述	62
函式 (Functions)	62
Call by value	63
Call by pointer	63
Call by reference	63
volatile 宣告	64
static 宣告	65
extern "C" { ... }	65
利基函式庫	67
函式庫介紹	67
I/O 功能腳位對應表	69
GPIO	72

指令詳細說明：	73
unsigned char <i>State</i> = InitialGpioState(unsigned char <i>GPIO</i> ;	73
void Low(unsigned char <i>GPIO</i> ;	73
void High(unsigned char <i>GPIO</i> ;	73
void Toggle(unsigned char <i>GPIO</i> ;	74
unsigned char <i>State</i> = In(unsigned char <i>GPIO</i> ;	74
void Input(unsigned char <i>GPIO</i> ;	74
void Output(unsigned char <i>GPIO</i> ;	74
unsigned short <i>State</i> = ReadPort(unsigned char <i>Port</i> ;	75
void WritePort(unsigned char <i>Port</i> , unsigned short <i>Value</i> ;	75
void SetDirPort(unsigned char <i>Port</i> , unsigned short <i>Value</i> ;	75
Event 相關設定：	76
unsigned char <i>State</i> = SetGpioEvent(unsigned char <i>EventNo</i> ,	76
unsigned char <i>State</i> = SetGpioEventOff(unsigned char <i>EventNo</i> ;	77
void GpioEvent0(void);	77
ADC	78
指令詳細說明：	78
unsigned char <i>State</i> = SetAdc(unsigned char <i>Channel</i> ;	78
unsigned short <i>State</i> = GetAdc(unsigned char <i>Channel</i> ;	78
unsigned char <i>State</i> = SetAdcOff(unsigned char <i>Channel</i> ;	79
Timer 0	80
產生固定時間 Event 和計數功能	80
unsigned char <i>State</i> = SetTm0(unsigned long <i>MaxCount</i> ;	80
unsigned char <i>State</i> = SetTm0Value(unsigned long <i>TmValue</i> ;	80
unsigned char <i>State</i> = GetTm0Value(unsigned long <i>TmValue</i> ;	81
GetTm0Value(<i>TmValue</i>);	81
unsigned char <i>State</i> = SetTm0Off(void);	81
Event 相關設定	81
unsigned char <i>State</i> = SetTm0Event(unsigned char <i>UpdateEvent</i> ;	81
void Tm0UpdateEvent(void);	81
Timer 1	83
產生固定時間 Event 和計數功能	83
unsigned char <i>State</i> = SetTm1(unsigned long <i>MaxCount</i> ;	83
unsigned char <i>State</i> = SetTm1Value(unsigned long <i>TmValue</i> ;	83
unsigned char <i>State</i> = GetTm1Value(unsigned long <i>TmValue</i> ;	84
unsigned char <i>State</i> = SetTm1Off(void);	84
Event 相關設定	84
unsigned char <i>State</i> = SetTm1Event(unsigned char <i>UpdateEvent</i> ;	84

void Tm1UpdateEvent(void);.....	85
Timer 2.....	86
產生固定時間 Event 和計數功能.....	87
unsigned char <i>State</i> = SetTm2Counter(unsigned char <i>InputMode</i> ,.....	87
unsigned char <i>State</i> = SetTm2CounterValue(unsigned short <i>TmValue</i>);.....	88
unsigned char <i>State</i> = GetTm2CounterValue(unsigned short <i>TmValue</i>);.....	88
unsigned char <i>State</i> = SetTm2CounterOff(void);	88
輸出 PWM 訊號功能.....	88
unsigned char <i>State</i> = SetTm2Pwm(unsigned short <i>Prescaler</i> ,	89
unsigned char <i>State</i> = SetTm2PwmCh(unsigned char <i>Channel</i> ,.....	89
unsigned char <i>State</i> = SetTm2PwmOff(void)	90
單一輸入脈波寬度量測功能.....	90
unsigned char <i>State</i> = SetTm2PulseIn(unsigned char <i>Input</i> ,.....	90
unsigned char <i>State</i> = GetTm2PulseInValue(unsigned Short <i>Value</i>);.....	91
unsigned char <i>State</i> = SetTm2PulseInOff(void)	91
單一脈波輸出功能	91
unsigned char <i>State</i> = SetTm2PulseOut(unsigned char <i>Output</i> ,.....	91
unsigned char <i>State</i> = GetTm2PulseOutState(void);.....	92
unsigned char <i>State</i> = SetTm2PulseOutOff(void)	93
頻率輸出功能	93
unsigned char <i>State</i> = SetTm2FreqOut(unsigned char <i>Output</i> ,.....	93
unsigned char <i>State</i> = SetTm2FreqOutOff(void);	93
編碼器解碼功能.....	94
unsigned char <i>State</i> = SetTm2Decoder(unsigned char <i>Input</i> ,	
unsigned short <i>Prescaler</i> ,.....	94
unsigned char <i>State</i> = SetTm2DecoderValue(short <i>TmValue</i>);	94
unsigned char <i>State</i> = GetTm2DecoderValue(short <i>TmValue</i>);.....	94
unsigned char <i>State</i> = SetTm2DecoderOff(void);	95
Event 相關設定.....	95
unsigned char <i>State</i> = SetTm2Event(unsigned char <i>UpdateEvent</i>);	95
void Tm2UpdateEvent(void);.....	95
Timer 3.....	96
產生固定時間 Event 和計數功能.....	97
unsigned char <i>State</i> = SetTm3Counter(unsigned char <i>InputMode</i> ,.....	97
unsigned char <i>State</i> = SetTm3CounterValue(unsigned short <i>TmValue</i>);.....	98
unsigned char <i>State</i> = GetTm3CounterValue(unsigned short <i>TmValue</i>);.....	98
unsigned char <i>State</i> = SetTm3CounterOff(void);	98
輸出 PWM 訊號功能.....	98

unsigned char <i>State</i> = SetTm3Pwm(unsigned short <i>Prescaler</i> ,	99
unsigned char <i>State</i> = SetTm3PwmCh(unsigned char <i>Channel</i> ,	99
unsigned char <i>State</i> = SetTm3PwmOff(void)	100
單一輸入脈波寬度量測功能	100
unsigned char <i>State</i> = SetTm3PulseIn(unsigned char <i>Input</i> ,	100
unsigned char <i>State</i> = GetTm3PulseInValue(unsigned Short <i>Value</i>);	101
unsigned char <i>State</i> = SetTm3PulseInOff(void)	101
單一脈波輸出功能	101
unsigned char <i>State</i> = SetTm3PulseOut(unsigned char <i>Output</i> ,	102
unsigned char <i>State</i> = GetTm3PulseOutState(void);	102
unsigned char <i>State</i> = SetTm3PulseOutOff(void)	103
頻率輸出功能	103
unsigned char <i>State</i> = SetTm3FreqOut(unsigned char <i>Output</i> ,	103
unsigned char <i>State</i> = SetTm3FreqOutOff(void);	104
編碼器解碼功能	104
unsigned char <i>State</i> = SetTm3Decoder(unsigned char <i>Input</i> ,	104
unsigned char <i>State</i> = SetTm3DecoderValue(short <i>TmValue</i>);	104
unsigned char <i>State</i> = GetTm3DecoderValue(short <i>TmValue</i>);	104
unsigned char <i>State</i> = SetTm3DecoderOff(void);	105
Event 相關設定	105
unsigned char <i>State</i> = SetTm3Event(unsigned char <i>UpdateEvent</i>);	105
void Tm3UpdateEvent(void);	105
Timer 4	106
產生固定時間 Event 和計數功能	107
unsigned char <i>State</i> = SetTm4Counter(unsigned char <i>InputMode</i> ,	107
unsigned char <i>State</i> = SetTm4CounterValue(unsigned short <i>TmValue</i>);	108
unsigned char <i>State</i> = GetTm4CounterValue(unsigned short <i>TmValue</i>);	108
unsigned char <i>State</i> = SetTm4CounterOff(void);	108
輸出 PWM 訊號功能	108
unsigned char <i>State</i> = SetTm4Pwm(unsigned short <i>Prescaler</i> ,	108
unsigned char <i>State</i> = SetTm4PwmCh(unsigned char <i>Channel</i> ,	109
unsigned char <i>State</i> = SetTm4PwmOff(void)	110
單一輸入脈波寬度量測功能	110
unsigned char <i>State</i> = SetTm4PulseIn(unsigned char <i>Input</i> ,	110
unsigned char <i>State</i> = GetTm4PulseInValue(unsigned Short <i>Value</i>);	111
unsigned char <i>State</i> = SetTm4PulseInOff(void)	111
單一脈波輸出功能	111
unsigned char <i>State</i> = SetTm4PulseOut(unsigned char <i>Output</i> ,	111

unsigned char <i>State</i> = GetTm4PulseOutState(void);.....	112
unsigned char <i>State</i> = SetTm4PulseOutOff(void)	112
頻率輸出功能	112
unsigned char <i>State</i> = SetTm4FreqOut(unsigned char <i>Output</i> ,.....	113
unsigned char <i>State</i> = SetTm4FreqOutOff(void);	113
編碼器解碼功能	113
unsigned char <i>State</i> = SetTm4Decoder(unsigned char <i>Input</i> ,	113
unsigned char <i>State</i> = SetTm4DecoderValue(short <i>TmValue</i>);	114
unsigned char <i>State</i> = GetTm4DecoderValue(short <i>TmValue</i>);.....	114
unsigned char <i>State</i> = SetTm4DecoderOff(void);	115
Event 相關設定	115
unsigned char <i>State</i> = SetTm4Event(unsigned char <i>UpdateEvent</i>);	115
void Tm4UpdateEvent(void);.....	115
RTC	116
指令詳細說明	116
unsigned char <i>State</i> = SetRtc(unsigned char <i>Prescaler</i> , unsigned long <i>Match</i>);	116
unsigned char <i>State</i> = GetRtcValue(unsigned long <i>Value</i>);.....	116
unsigned char <i>State</i> = GetRtcClk(void);	117
unsigned char <i>State</i> = SetRtcOff(void);	117
Event 相關設定	117
unsigned char <i>State</i> = SetRtcEvent(unsigned char <i>Match</i> , unsigned char <i>Clock</i>);.....	117
void RtcMatchEvent(void);	117
void RtcClkEvent(void);.....	118
OPA0/CMP0.....	119
指令詳細說明	119
unsigned char <i>State</i> = SetCmp0(unsigned char <i>Mode</i>);	119
unsigned char <i>State</i> = GetCmp0(void);	119
unsigned char <i>State</i> = SetCmp0Off(void);	119
Event 相關設定	119
unsigned char <i>State</i> = SetCmp0Event(unsigned char <i>Falling</i> ,	119
void Cmp0FallingEdgeEvent(void);	120
void Cmp0RisingEdgeEvent(void);	120
OPA1/CMP1.....	121
指令詳細說明	121
unsigned char <i>State</i> = SetCmp1(unsigned char <i>Mode</i>);	121
unsigned char <i>State</i> = GetCmp1(void);	121
unsigned char <i>State</i> = SetCmp1Off(void);	121

Event 相關設定	121
unsigned char <i>State</i> = SetCmp1Event(unsigned char <i>Falling</i> ,	121
void Cmp1FallingEdgeEvent(void);	122
void Cmp1RisingEdgeEvent(void);	122
EE-DATA	123
指令詳細說明	123
unsigned char <i>State</i> = WriteEEData(unsigned short <i>Address</i> , void * <i>Variable</i> , ..	123
unsigned char <i>State</i> = ReadEEData(unsigned short <i>Address</i> , void * <i>Variable</i> , ...	123
I2C0	125
指令詳細說明	125
unsigned char <i>State</i> = SetI2c0(unsigned char <i>Speed</i> ,	125
unsigned char <i>State</i> = I2c0MasterTransmit(void * <i>Array</i> ,	126
unsigned char <i>State</i> = I2c0MasterReceive(void * <i>Array</i> ,	126
unsigned char <i>State</i> = GetI2c0MasterState(void);	127
unsigned char <i>State</i> = GetI2c0SlaveState(void);	127
unsigned char <i>State</i> = I2c0SlaveTransmit(void * <i>Array</i> , unsigned short <i>Length</i>);	127
unsigned char <i>State</i> = I2c0SlaveReceive(void * <i>Array</i> , unsigned short <i>Length</i>);	127
unsigned char <i>State</i> = SetI2c0Off(void);	128
Event 相關設定	128
unsigned char <i>State</i> = SetI2c0Event(unsigned char <i>MasterEventEnable</i> ,	128
void I2c0MasterEvent(void);	129
void I2c0SlaveEvent(void);	129
I2C1	130
指令詳細說明	130
unsigned char <i>State</i> = SetI2c1(unsigned char <i>Speed</i> ,	130
unsigned char <i>State</i> = I2c1MasterTransmit(void * <i>Array</i> ,	131
unsigned char <i>State</i> = I2c1MasterReceive(void * <i>Array</i> ,	131
unsigned char <i>State</i> = GetI2c1MasterState(void);	132
unsigned char <i>State</i> = GetI2c1SlaveState(void);	132
unsigned char <i>State</i> = I2c1SlaveTransmit(void * <i>Array</i> , unsigned short <i>Length</i>);	132
unsigned char <i>State</i> = I2c1SlaveReceive(void * <i>Array</i> , unsigned short <i>Length</i>);	132
unsigned char <i>State</i> = SetI2c1Off(void);	132
Event 相關設定	132
unsigned char <i>State</i> = SetI2c1Event(unsigned char <i>MasterEventEnable</i> ,	132
void I2c1MasterEvent(void);	133
void I2c1SlaveEvent(void);	133
SPIO	135
指令詳細說明	135

unsigned char <i>State</i> = SetSpi0(unsigned short <i>ClkPrescaler</i> ,	135
unsigned char <i>State</i> = Spi0MasterStart(void);	138
unsigned char <i>State</i> = Spi0MasterTransmit(void * <i>Array</i> , unsigned short <i>Length</i>);	138
unsigned char <i>State</i> = Spi0MasterReceive(void * <i>RxArray</i> ,	138
unsigned char <i>State</i> = Spi0MasterStop(void);	139
unsigned char <i>State</i> = GetSpi0MasterState(void);	139
unsigned char <i>State</i> = Spi0SlaveReceive(void * <i>Array</i> , unsigned short <i>Length</i>);	139
unsigned char <i>State</i> = Spi0SlaveTransmit(void * <i>Array</i> , unsigned short <i>Length</i>);	140
unsigned char <i>State</i> = GetSpi0SlaveState(unsigned char <i>RxFifoLevel</i>);	140
unsigned char <i>State</i> = ResetSpi0SlaveRxFifo(void);	140
unsigned char <i>State</i> = SetSpi0Off(void);	140
Event 相關設定	140
unsigned char <i>State</i> = SetSpi0Event(unsigned char <i>MasterEvent</i> ,	140
void Spi0MasterEvent(void);	140
void Spi0SlaveEvent(void);	141
SPI1	142
指令詳細說明	142
unsigned char <i>State</i> = SetSpi1(unsigned short <i>ClkPrescaler</i> ,	142
unsigned char <i>State</i> = Spi1MasterStart(void);	145
unsigned char <i>State</i> = Spi1MasterTransmit(void * <i>Array</i> , unsigned short <i>Length</i>);	146
unsigned char <i>State</i> = Spi1MasterReceive(void * <i>RxArray</i> ,	146
unsigned char <i>State</i> = Spi1MasterStop(void);	146
unsigned char <i>State</i> = GetSpi1MasterState(void);	146
unsigned char <i>State</i> = Spi1SlaveReceive(void * <i>Array</i> , unsigned short <i>Length</i>);	147
unsigned char <i>State</i> = Spi1SlaveTransmit(void * <i>Array</i> , unsigned short <i>Length</i>);	147
unsigned char <i>State</i> = GetSpi1SlaveState(unsigned char <i>RxFifoLevel</i>);	147
unsigned char <i>State</i> = ResetSpi1SlaveRxFifo(void);	147
unsigned char <i>State</i> = SetSpi1Off(void);	147
Event 相關設定	147
unsigned char <i>State</i> = SetSpi1Event(unsigned char <i>MasterEvent</i> ,	147
void Spi1MasterEvent(void);	148
void Spi1SlaveEvent(void);	148
UART0	149
指令詳細說明 :	149
unsigned char <i>State</i> = SetUart0(unsigned long <i>Baudrate</i> ,	149
unsigned char <i>State</i> = SendUart0Data(void * <i>Array</i> , unsigned short <i>Length</i>); ..	150

unsigned char <i>State</i> = GetUart0TxState(void);	151
unsigned short <i>State</i> = GetUart0Data(void * <i>Array</i> , unsigned short <i>Length</i>);....	151
unsigned char <i>State</i> = GetUart0RxState(unsigned char <i>RxFifoLevel</i>);	151
unsigned char <i>State</i> = ResetUart0RxFifo(void);	152
unsigned char <i>State</i> = SetUart0Off(void);	152
Event 相關設定	152
unsigned char <i>State</i> = SetUart0Event(unsigned char <i>Uart0TxEvent</i> ,	152
void Uart0TxEvent (void);	152
void Uart0RxEvent (void);	152
UART1	154
指令詳細說明 :	154
unsigned char <i>State</i> = SetUart1(unsigned long <i>Baudrate</i> ,	154
unsigned char <i>State</i> = SendUart1Data(void * <i>Array</i> , unsigned short <i>Length</i>); ..	156
unsigned char <i>State</i> = GetUart1TxState(void);	156
unsigned short <i>State</i> = GetUart1Data(void * <i>Array</i> , unsigned short <i>Length</i>);....	156
unsigned char <i>State</i> = GetUart1RxState(unsigned char <i>RxFifoLevel</i>);	156
unsigned char <i>State</i> = ResetUart1RxFifo(void);	156
unsigned char <i>State</i> = SetUart1Off(void);	157
Event 相關設定	157
unsigned char <i>State</i> = SetUart1Event(unsigned char <i>Uart1TxEvent</i> ,	157
void Uart1TxEvent (void);	157
void Uart1RxEvent (void);	157
USB	158
指令詳細說明 :	158
unsigned char <i>State</i> = SendUsbData(void * <i>Array</i> , unsigned short <i>Length</i>);....	159
unsigned char <i>State</i> = GetUsbTxState(void);	159
unsigned char <i>State</i> = GetUsbRxState(unsigned short <i>RxBufferLevel</i>);	159
unsigned char <i>State</i> = GetUsbData(void * <i>Array</i> , unsigned short <i>Length</i>);	159
unsigned char <i>State</i> = ResetUsbRxBuffer(void);	160
unsigned char <i>State</i> = SetUsbOff(void);	160
Event 相關設定	160
unsigned char <i>State</i> = SetUsbEvent(unsigned char <i>UsbTxEvent</i> ,	160
void UsbTxEvent (void);	160
void UsbRxEvent (void);	160
Power Saving	161
指令詳細說明 :	161
void Sleep(unsigned char <i>Mode</i>);	161
unsigned char <i>State</i> = GetResetState(void);	162

Other commands	163
指令詳細說明 :	164
void Pause(unsigned short <i>Time</i>);	164
unsigned char <i>State</i> = SetMainClk(unsigned char <i>Mode</i>);	164
unsigned char <i>State</i> = SetX32Clk(unsigned char <i>Mode</i>);	164
unsigned char <i>State</i> = SetIdeOff(void);	164
unsigned char <i>State</i> = GlobalEventControl(unsigned char <i>Enable</i>);	164
void GetLibVer(char * <i>Version</i>);	164
void GetSysTick(unsigned long <i>TickValue</i>);	164
unsigned char <i>State</i> = SetClkOut(unsigned char <i>Mode</i>);	165
CMDBUS™ 模組物件程式庫應用	166
硬體接線	166
軟體物件呼叫	166

前言

歡迎

我們很高興您使用利基的產品。無論您是一個微控制器的初學者，或是在此領域已經有豐富經驗的老手，我們確信您將可立即體驗本系統帶您進入電子世界的樂趣。本手冊將提供一些使用本產品需具備的技術和經驗的相關資訊。對一個微控制器的新手而言，當然最好具備有基本的電子背景及C語言知識，如此將有助於閱讀本手冊。但無論如何，不管您的興趣如何，也不論您具備怎樣的電子背景，利基團隊均真誠歡迎您愉快地進入微控制器的奇妙世界。

利基以獨特的技術，提供各式功能強大的模組，讓您以最短的時間及最省力的方式來開發您的硬體應用。例如：在您的模型越野車加個遙控相機控制功能、或在您的房子或自製的機器人上加個精緻的自動照明控制系統等等。可能您正對學習微控制器發生興趣，或者您可能是一個高中老師，且您正想要將微控制器引進學校的課程。是的，本產品內含一個工業標準的高品質微控制器核心，所有的應用將有無限的可能，唯一不同的是您的創意和經驗了。不論您的興趣如何，我們確定本系統不僅可以完成您獨特的創意或發明，重要的是，它可以讓您真實地獲得學習使用微控制器的經驗。

微控制器無所不在

打從您每天早上打點第一杯咖啡及土司麵包，一直到晚上上床前設定的鬧鐘，各種不同的電子家電，讓您的每一天均充滿著微控制器的影子以及它所帶來的便利。

每一部汽車可能內含許多微控制器，從一般的空調系統到精密的引擎管理系統，均必須使用到微控制器，儀表板上的各種儀器，也必須靠微控制器來指示哪個是好的或哪個有問題。您看過電視或 DVD 嗎？毫無疑問地，這些家電和它們的遙控器，也都內含有微控制器。以前可能是工業用或實驗室研究用的電子裝置，現在已經應用在我們的生活中了。所以，既然微控制器如此好用且無所不在，那我們何不拿來用在我們自己的創意作品上呢？現在，您可以使用本系統來實現了。可能您想在您的火車模型上加裝自動控制系統，或在您的房子裝個保全系統，或甚至一個獨一無二的特殊作品。所有的一切端看您的想像力與創造了！

利基的平台

微控制器專案的開發可以從許多不同的角度來達成。傳統的方式是以低階的組合語言撰寫軟體，或者花很長的時間，學習用不甚方便的高階語言；此外，還必須針對不同的硬體，設計不同的介面，將您的特殊應用和微控制器做連結，這些硬體可能包括液晶顯示器、開關、發光二極體等等。這將很耗時間，而且可能只適合工業界的特殊設計及量產產品。有鑑於此，

利基利用基於現今世界上最普遍也最強大的嵌入式開發平台 ARM 系統，其所提供的最新高效能單晶片控制器 Cortex M3，打造了一既為新手入門亦為老經驗工程師設計之開發系統 Arminno™。

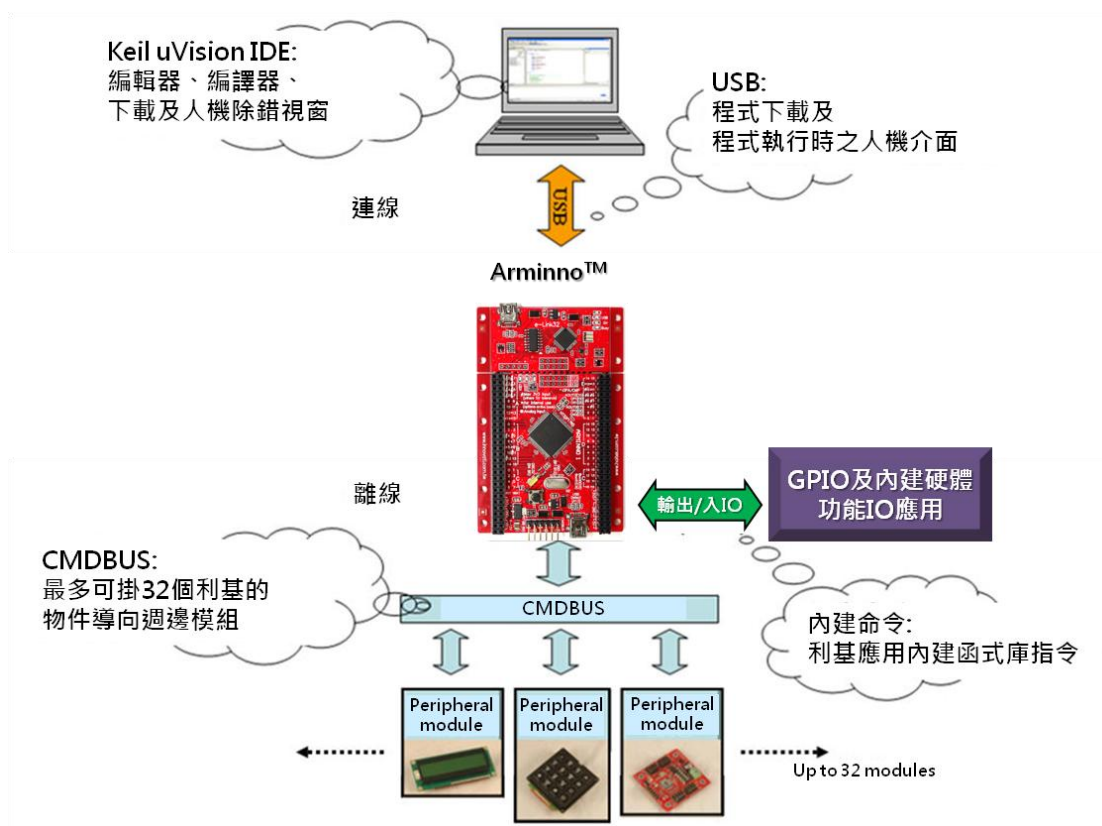
Arminno™可讓使用者在熟悉的 Keil 編輯器上，以 C++ 語言，加上利基為 Arminno™所獨家量身打造之使用簡單卻又功能強大之 Library 函式庫設定，以及相關之硬體搭配，讓您以最容易的方式，來完成較複雜的功能。另外，透過內建指令，Arminno™可以經由特殊設計的 CMDBUS™介面，來和利基提供的各種模組結合，這些模組是一些已經結合軟硬體的功能模組，複雜的硬體控制被單純的軟體方法取代了。利用這種方法，將大幅減少您實現創意應用所需花的時間和精力。如此一來，微控制器的使用將不再是專業人士的專利，所有的人都可以輕易的寫程式來控制它了！

系統概述

簡介

本系統包含了幾個部份：Keil μ Vision IDE 是一個安裝在電腦上的軟體平臺，提供使用者以 C++ 程式語言編輯、編譯及下載程式；完成的程式碼透過 USB 介面，經 Holtek e-Link32 下載到 Arminno™ 單板電腦 (Single Board Computer)，這也是系統的核心部份。此外 Keil μ Vision IDE 亦為一程式執行時的人機介面及除錯平台。

Arminno™ 提供三種資源：第一個是通用的 I/O；第二個是 CMDBUS™，透過它可同時並聯 32 個利基的週邊模組。第三個是 DEBUG 介面，用來讓 Keil μ Vision 終端視窗對 Arminno™ 上傳或下傳資料，DEBUG 介面不僅作為除錯使用，它同時也是一個方便的人機介面。



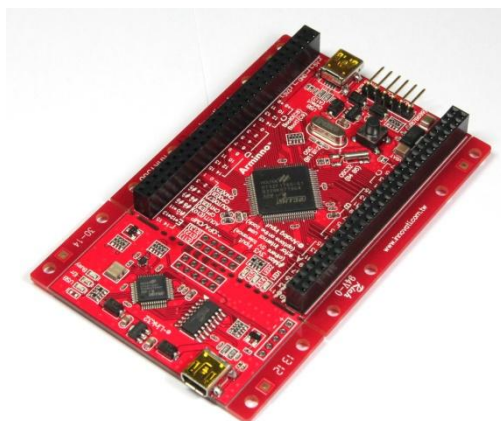
C++ 程式語言

每種語言的發展都有其開發背景，其中 C++ 是當今廣泛使用用於系統開發程式語言之一。它具有效率高、易於維護、可攜性及高度結構化之優點。利基的 Arminno™ 系統使用 C++ 高階程式語言。其強大功能除了深受有經驗的使用者喜愛，同時也兼顧初學者學習使

用。引導大家進入有趣的微控制器的世界。

Arminno™單板電腦

Arminno™為本系統的核心，它是一個小型的單板電腦(Single-Board Computer)，採用盛群半導體(HOLTEK)以 Cortex M3 為核心所生產之 HT1765 高速微處理器，Cortex M3 為 ARM 嵌入式系統的最新核心之一。在程式編寫過程的除錯階段，或者程式碼的下載時，Arminno™透過 Serial Wire Debug Port 連接 Holtek e-Link32，再經由 USB 線與 PC 連接，方便它與 PC 間的資料傳遞，但當程式下載完畢，它也可以脫離 PC 而獨立作業。



Arminno™內建硬體功能及相關函式庫支援

Arminno™內建多種硬體加速及擴充支援功能，例如 Timer、RTC、UART、I2C、SPI、EEDATA...等，擁有這些功能不僅可讓 Arminno™可加速相關指令執行，更可在同時間進行多種不同程序處理。利基在此一基礎上進一步開發了 Arminno™內建功能相關函式庫，讓使用者可以跳過既繁瑣又複雜之底層函式開發，進而直接享受由 Arminno™ Library 所帶來之易於上手及快速開發之便利性。

CMDBUS™週邊模組及函式庫

CMDBUS™智慧週邊模組的使用是利基 Arminno™系統的特色之一。CMDBUS™匯流排為利基針對所有 CMDBUS™模組所開發之通用匯流排，可利用定義不同 Module ID 將所有模組串接於 CMDBUS™匯流排上。

利基為各式 CMDBUS™模組開發了一系列完整之應用功能函式庫，可讓所有 CMDBUS™模組在透過 Arminno™之下，發揮其各式不同功能。

CMDBUS™模組組包含有 I/O 擴充模組、液晶顯示模組、馬達驅動模組、電子羅盤模組...等等。這些模組內建一微處理器控制晶片，此晶片針對各模組上之特性元件進行複雜及繁瑣

的底層控制，使用者只需“使用”該模組的各項功能而不需在花精神“開發”該模組，因此完全節省了使用者寶貴的工程開發時間。

安裝與開始使用

簡介

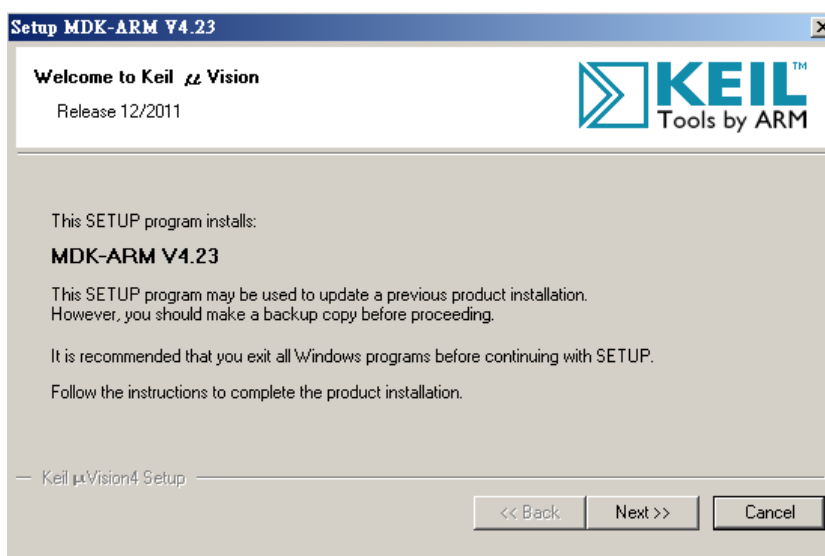
本章介紹如何安裝 Keil μ Vision, e-Link32 USB Driver 及 e-Link32 Keil Plugin 軟體。
本系統適用於 IBM 或與其相容的個人電腦，Windows XP 或以上的版本 Vista/Window7
等作業系統均可使用。電腦須有 USB 1.1 /2.0 接頭，用來下載程式碼及除錯。

安裝 Keil μ Vision IDE 開發環境

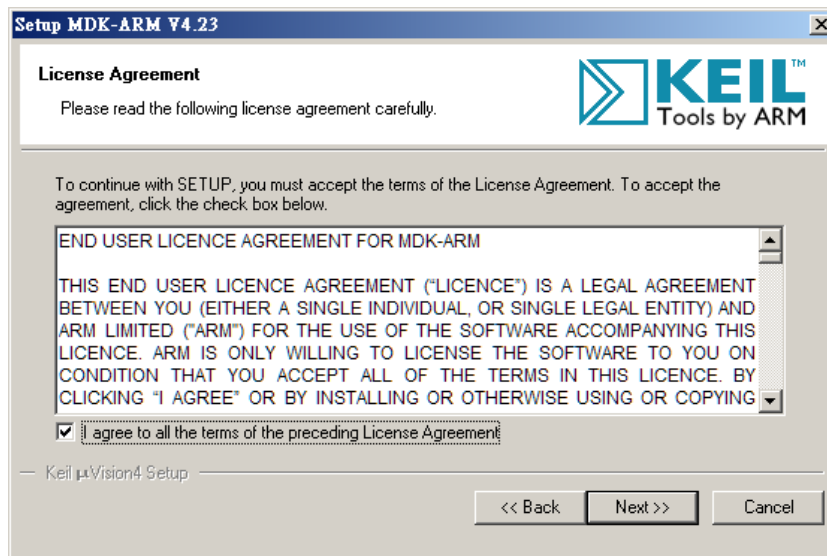
建議安裝版本 Keil μ Vision IDE V4.23 或以上(註)，請至
<https://www.keil.com/download/product/> 下載免費 MDK-ARM 32K 版本，以下說明
將以 4.23 版為例。注意,請以 Windows 的管理者(Administrator)權限來進行安裝。
MDK423 下載完成後，雙點擊圖示後開始執行安裝。



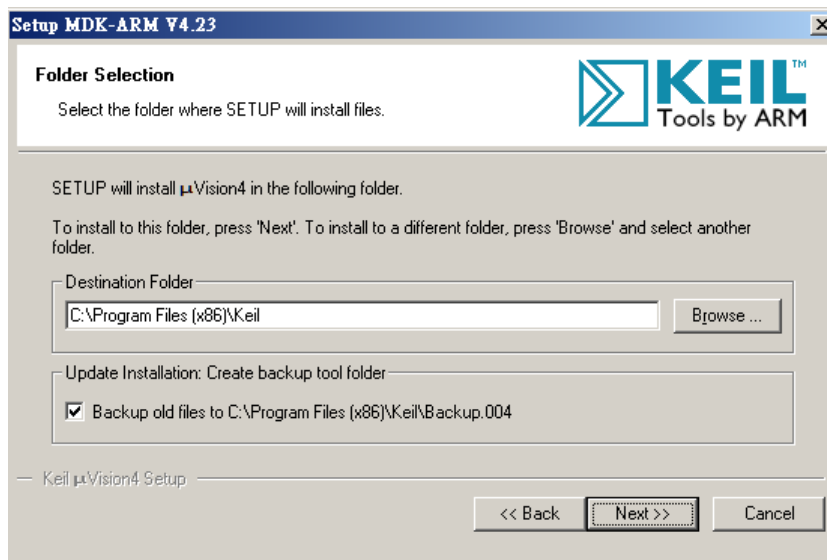
出現安裝對話框，按下一步(Next>>)開始安裝。



出現授權協議書對話框。閱讀授權事項，確定後在 agree to all the terms of the preceding License Agreement 核對框中打勾，並按下一步(Next>>)。



出現選擇安裝目錄對話框，如不需要更改安裝目錄，請直接按下一步(Next>>)。



出現使用者資訊對話框，請填入自己的名(First Name)，姓(Last Name)，公司名稱 (Company Name) 和電子郵件(E-mail)後按下一步(Next>>)繼續安裝。

Setup MDK-ARM V4.23

Customer Information
Please enter your information.

Please enter your name, the name of the company for whom you work and your E-mail address.

First Name:

Last Name:

Company Name:

E-mail:

— Keil μ Vision4 Setup —

<< Back Next >> Cancel

安裝程式開始執行安裝，檔案安裝完成，按下一步(Next>>)繼續安裝。

Setup MDK-ARM V4.23

Setup Status

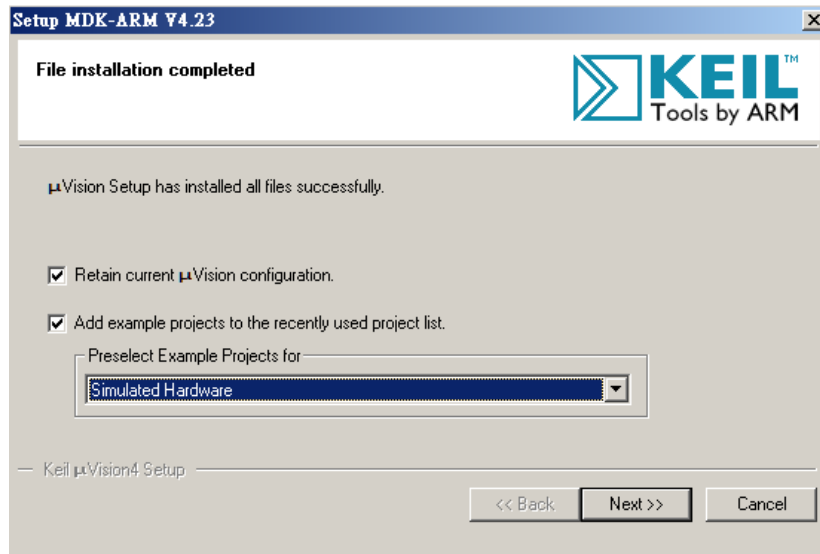
μ Vision Setup is performing the requested operations.

Install Files ...

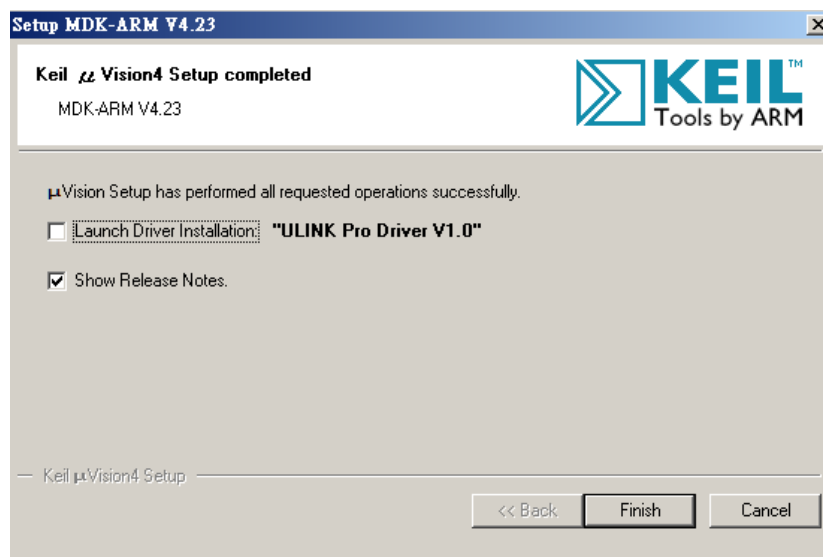
Installing TARMCM1.dll.

— Keil μ Vision4 Setup —

<< Back Next >> Cancel

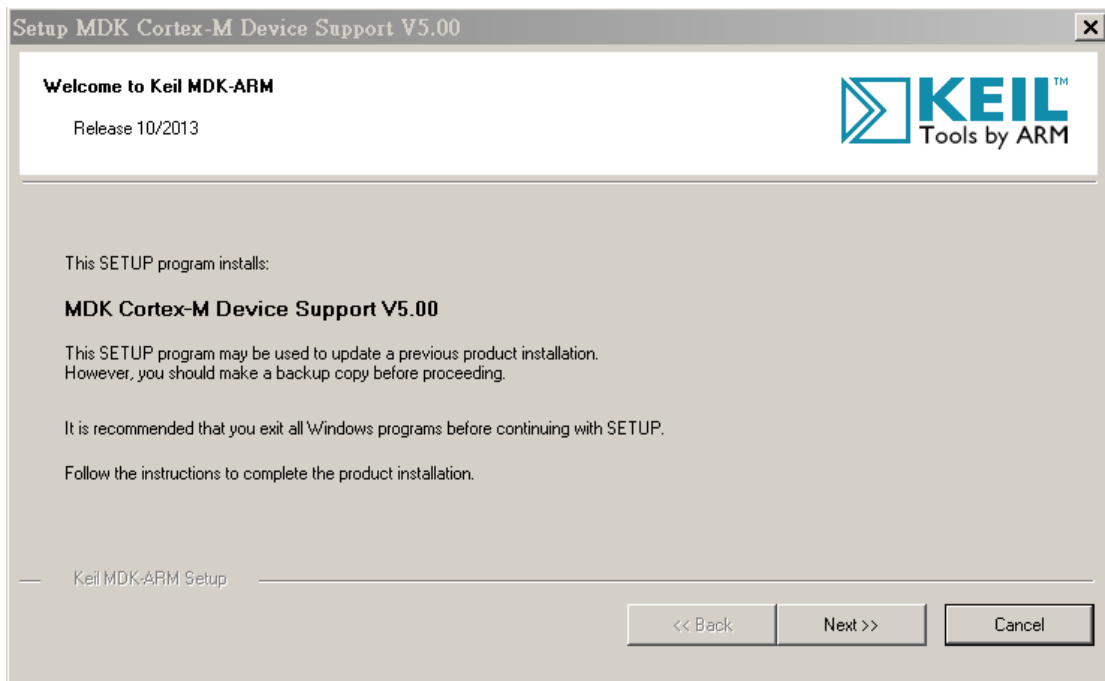


出現安裝完成對話框，因為不使用 ULINK Pro，所以請將 ULINK Pro Driver V1.0 打勾取消並按完成(Finish)。至此，Keil µVision IDE 開發環境已完成安裝。



註：建議使用 Keil 4.x 的版本來開發程式。但如果你有其他需求必須使用的 Keil 5.0 以上的版本(含)，因為目前暫不支援 Holtek IC，除了安裝 Keil 5.0 外，必須另外安裝 Legacy Support for Cortex-M Devices。請至 <http://www2.keil.com/mdk5/legacy> 下載並安裝。Keil 4.x 版本則請略過此步驟

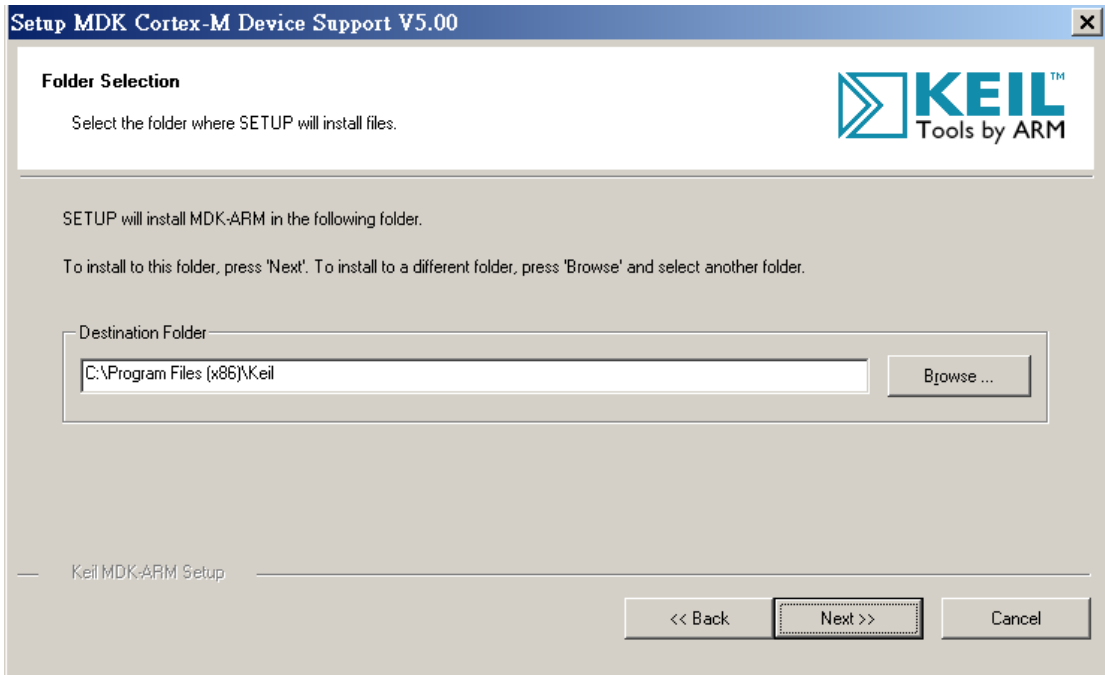
下載完成後，執行 Legacy Support for Cortex-M Devices, 按 Next 下一步



出現授權協議書對話框。閱讀授權事項，確定後在 I agree to all the terms of the preceding License Agreement 核對框中打勾，並按下一步(Next>>)

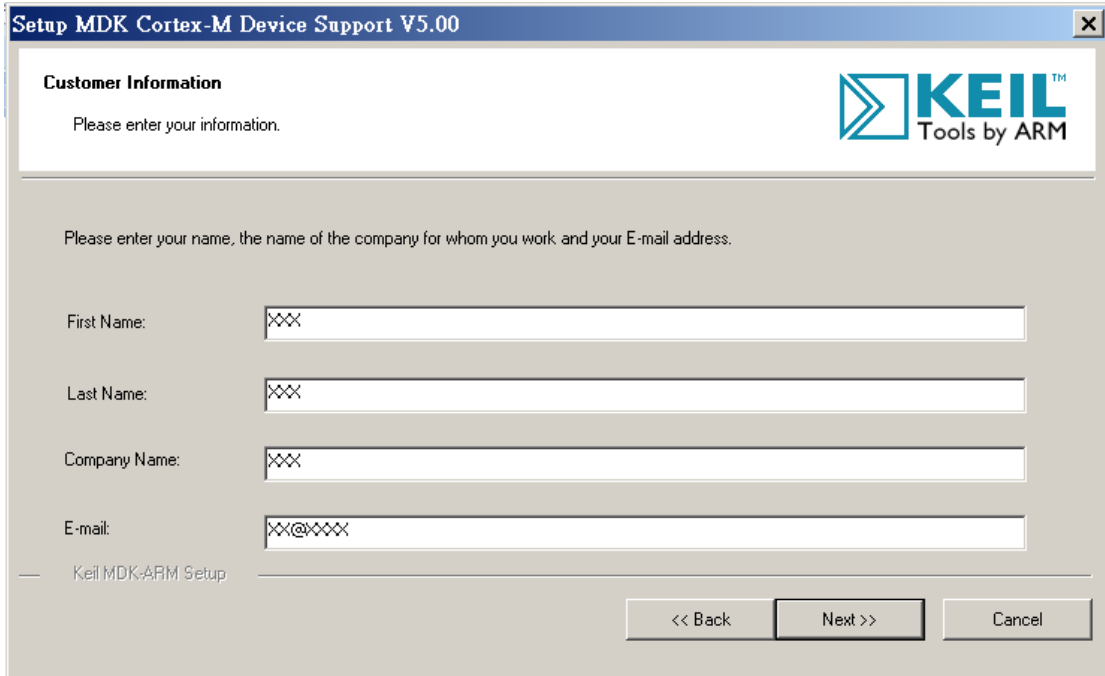


安裝到剛剛安裝 KEIL 5.0 的同目錄, 按下一步



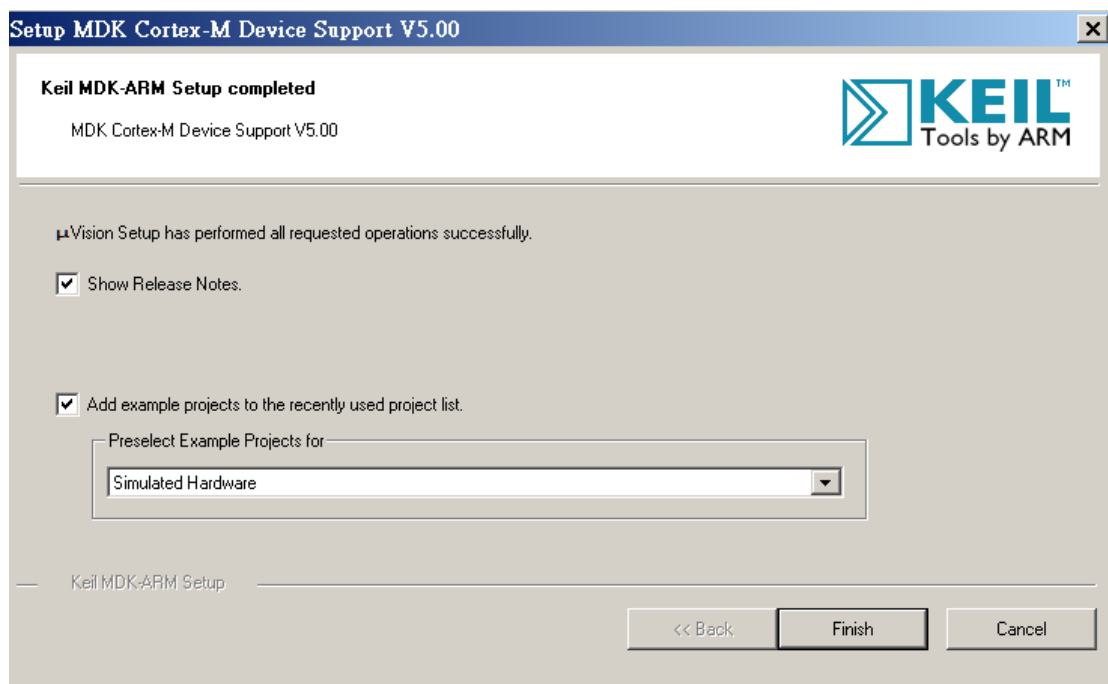
The screenshot shows the 'Folder Selection' window of the 'Setup MDK Cortex-M Device Support V5.00' installer. The window has a blue title bar and a grey background. In the top right corner is the KEIL Tools by ARM logo. The main text area contains instructions: 'Select the folder where SETUP will install files.', 'SETUP will install MDK-ARM in the following folder.', and 'To install to this folder, press 'Next'. To install to a different folder, press 'Browse' and select another folder.' Below this is a 'Destination Folder' section with a text box containing 'C:\Program Files (x86)\Keil' and a 'Browse ...' button. At the bottom, there are three buttons: '<< Back', 'Next >>' (which is highlighted with a black border), and 'Cancel'. A progress bar at the bottom left shows the current step is 'Keil MDK-ARM Setup'.

輸入個人資料後, 按 Next 下一步



The screenshot shows the 'Customer Information' window of the 'Setup MDK Cortex-M Device Support V5.00' installer. The window has a blue title bar and a grey background. In the top right corner is the KEIL Tools by ARM logo. The main text area contains instructions: 'Please enter your information.' and 'Please enter your name, the name of the company for whom you work and your E-mail address.' Below this are four input fields: 'First Name:', 'Last Name:', 'Company Name:', and 'E-mail:'. Each field has a placeholder text 'xxx' or 'xx@xxx'. At the bottom, there are three buttons: '<< Back', 'Next >>' (which is highlighted with a black border), and 'Cancel'. A progress bar at the bottom left shows the current step is 'Keil MDK-ARM Setup'.

按下 Finish 完成

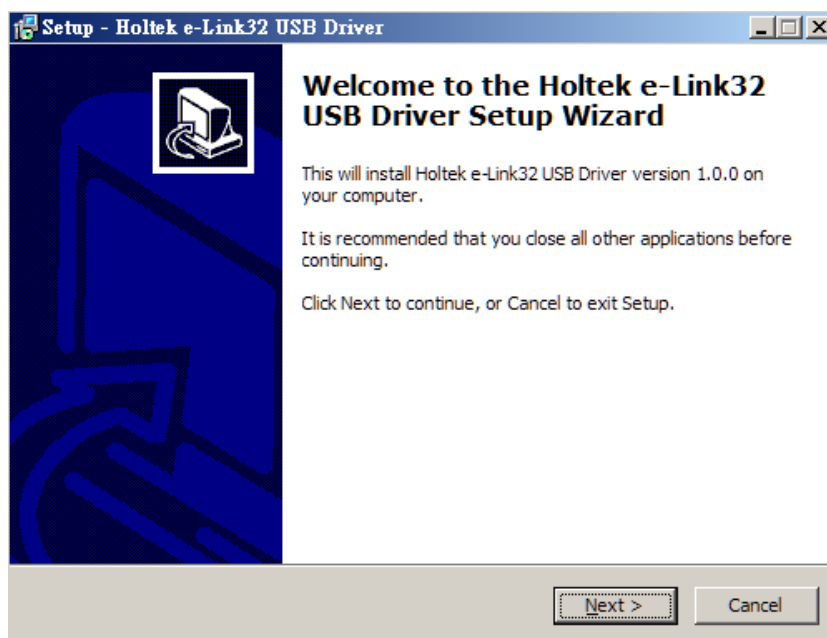


安裝 e-Link32 USB Driver

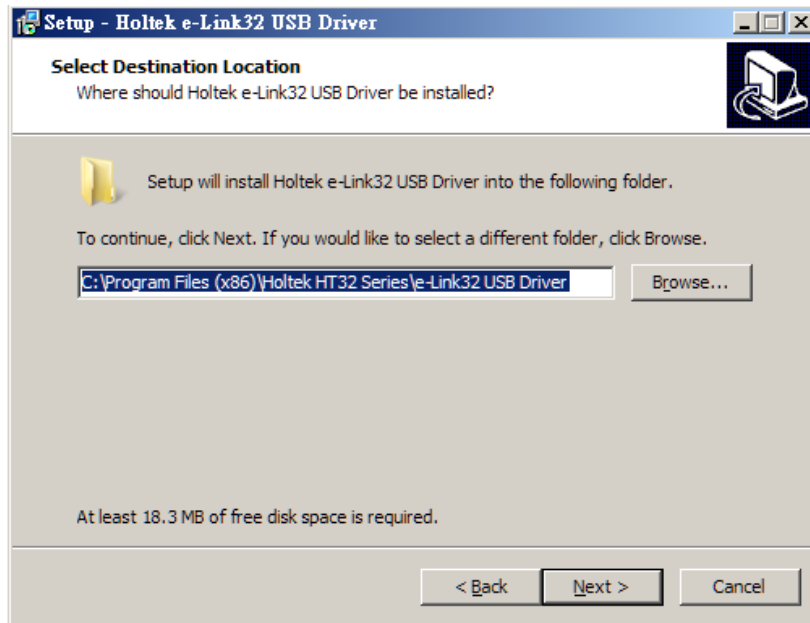
在未安裝 e-Link32 USB Driver 之前，請先不要把 Holtek e-Link32 連上 PC。安裝時請以 Windows 的管理者(Administrator)權限來進行安裝。請雙點擊 e-Link32 USB Driver e-Link32_USB_Driver_vXXX.exe 圖示後開始執行安裝。



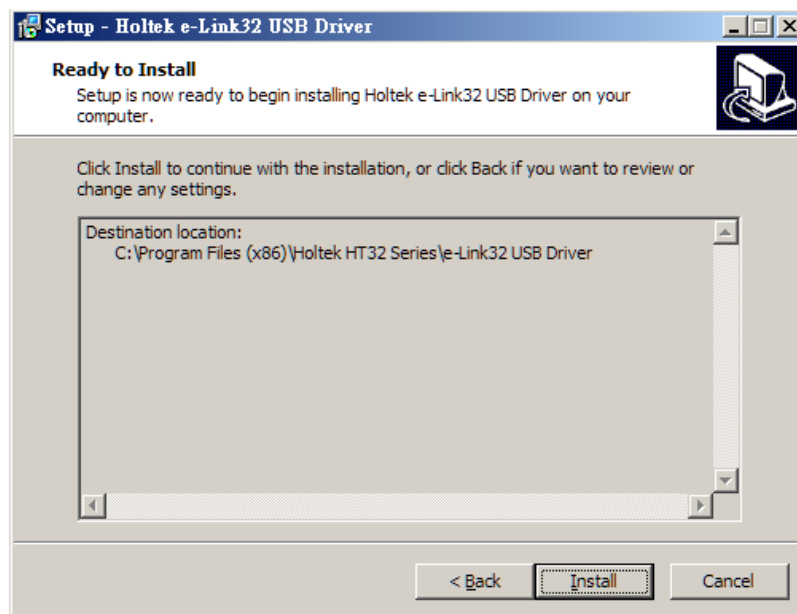
出現安裝對話框，請按下一步(Next>)開始安裝。

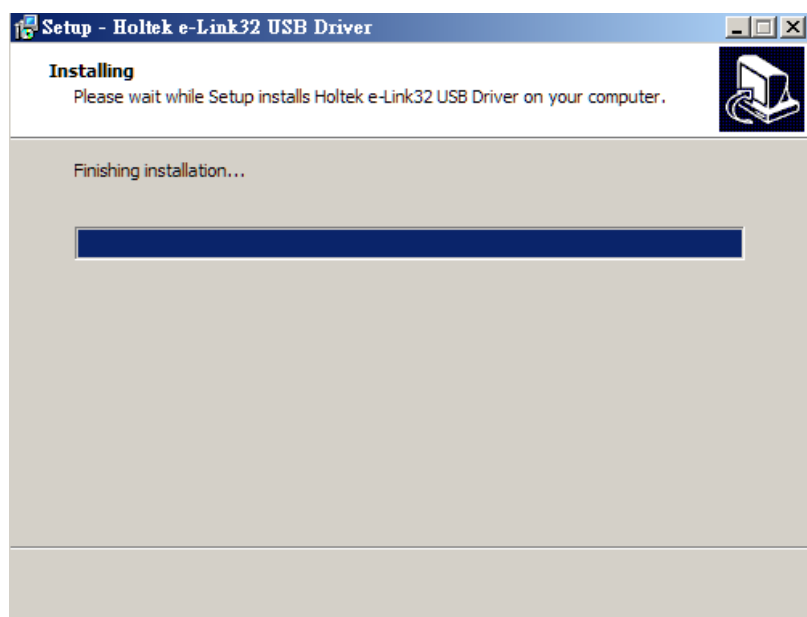


出現選擇安裝目錄對話框。如不需要更改安裝目錄，請直接按下一步(Next>)繼續安裝。

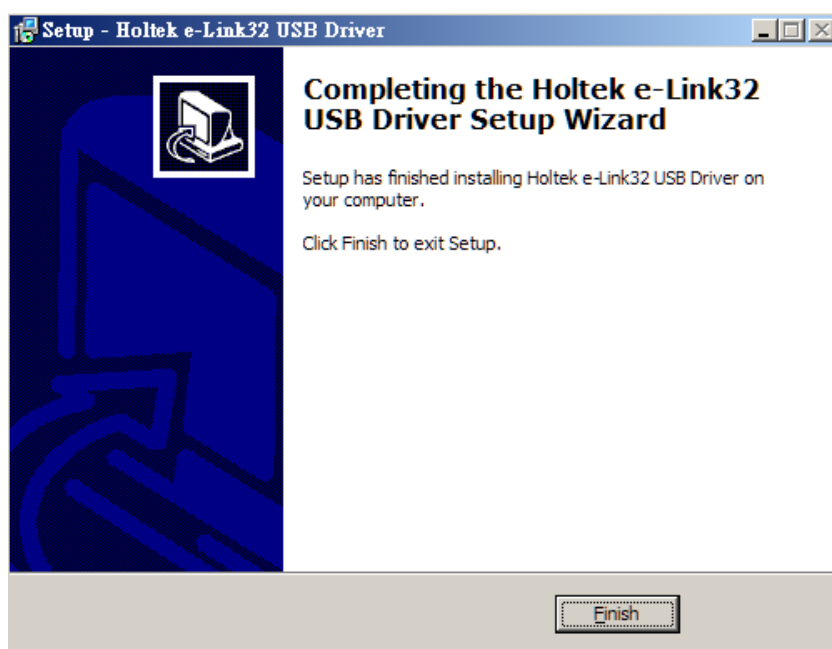


出現對話框，請按(Install)開始安裝。

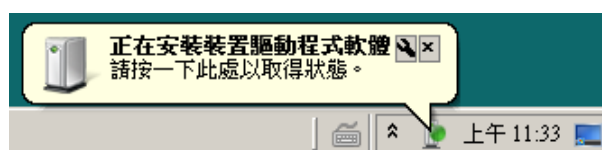




出現安裝完成對話框，請按完成(Finish)結束安裝。



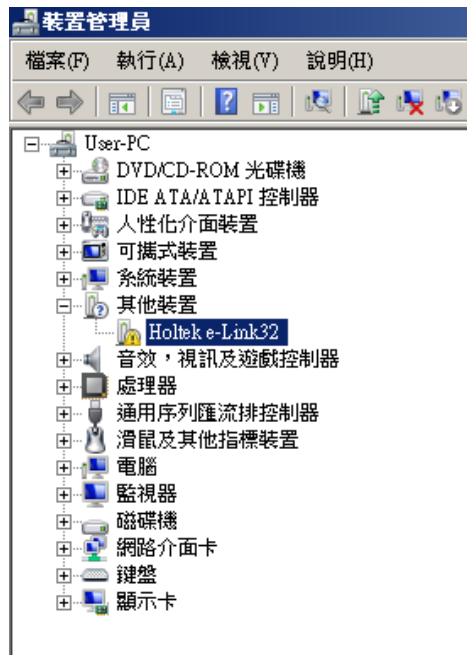
將連接 PC 的 USB Cable 插入 Holtek e-Link32 的 mini USB 接頭中，系統會自動開始尋找驅動程式軟體，並進行安裝。



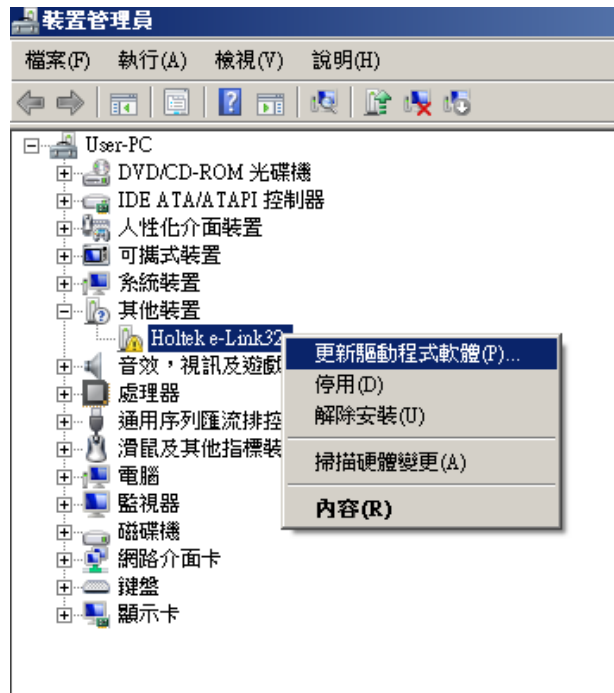
當 e-Link32 驅動程式軟體安裝成功則會顯示安裝成功訊息如下。



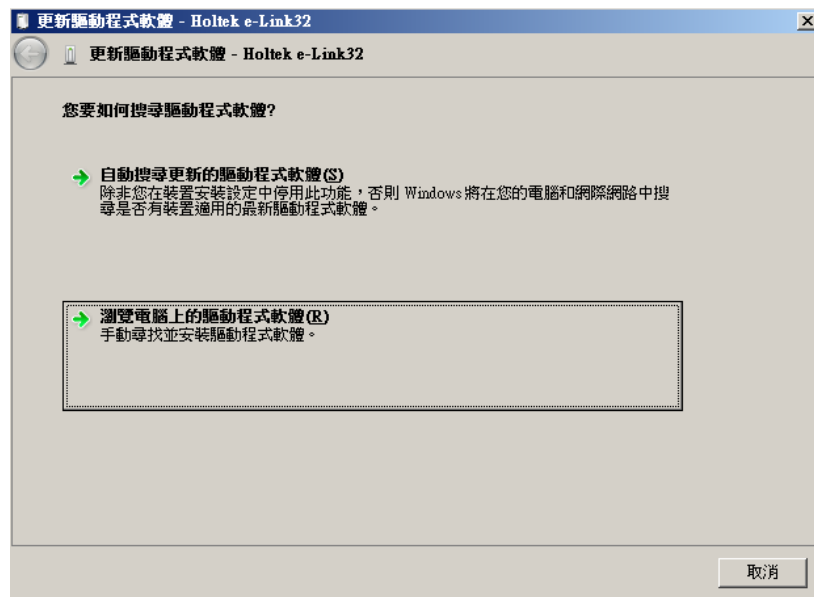
如果安裝失敗，請開啟開始-控制台-裝置管理員，在裝置管理員中找到未安裝成功的 Holtek e-Link32 裝置。



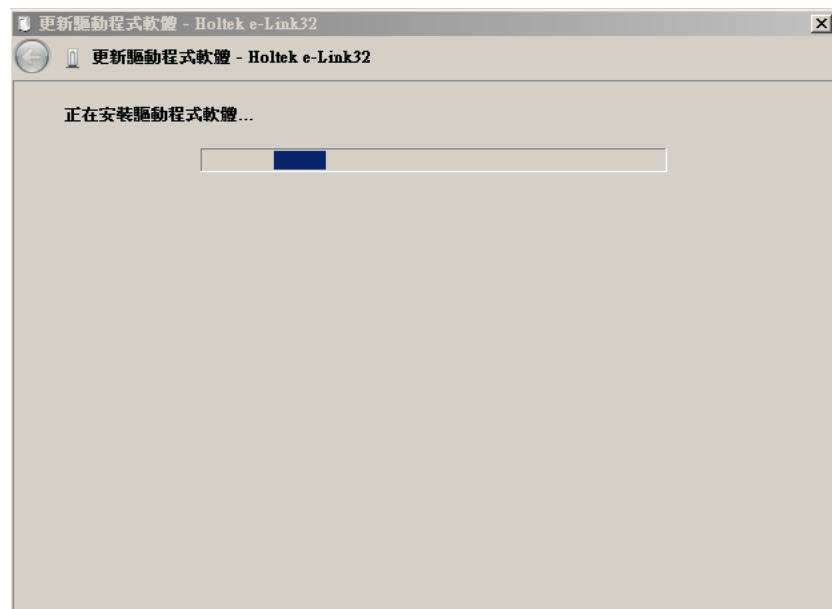
在 Holtek e-Link32 上按滑鼠右鍵，出現選單，選擇**更新驅動程式軟體**。



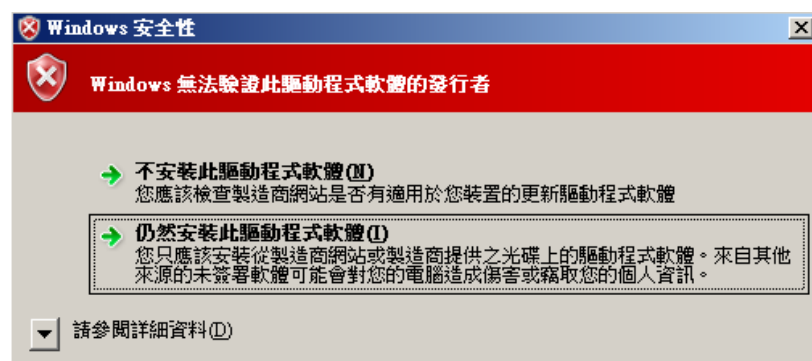
出現更新驅動程式軟體對話框，選擇瀏覽電腦上的驅動程式軟體。



出現更新驅動程式軟體對話框，按下瀏覽，選擇之前 Holtek e-Link32 中的安裝路徑後按下一步(N)。



安裝過程中如出現安全性警告畫面，請選擇仍然安裝此驅動程式軟體。



安裝完成！請按關閉後離開。



Win8/8.1 單次開機停用數位簽章辦法

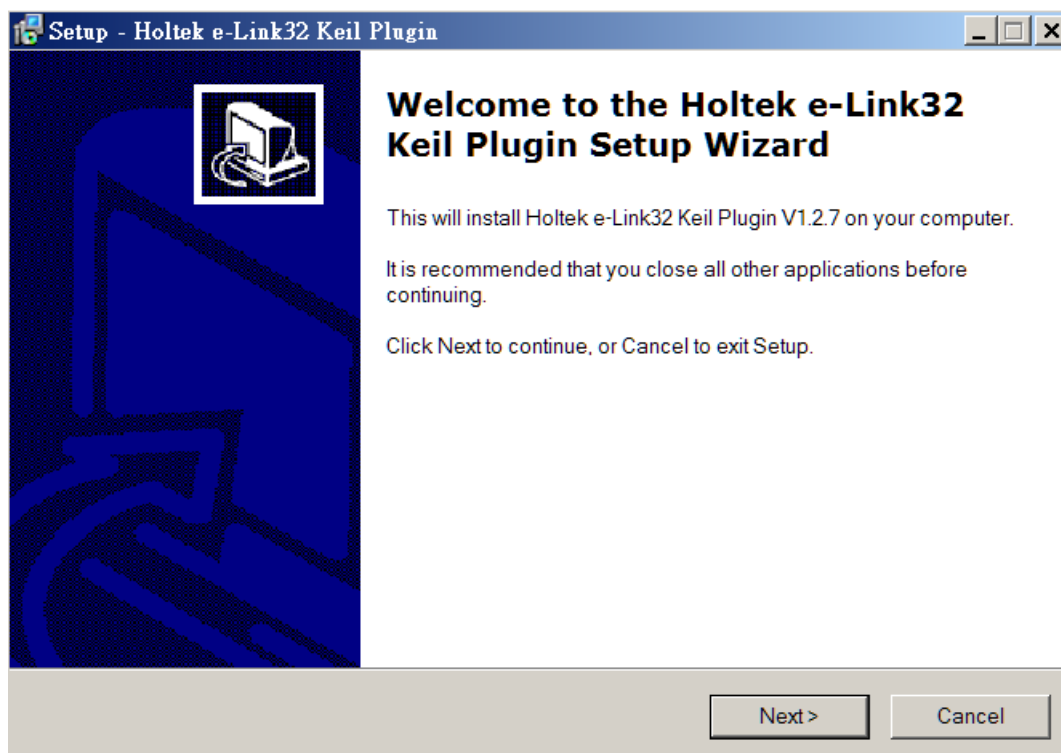
已知 e-Link32 的驅動程式在 Win8/8.1 中會有因數位簽章而無法使用的問題，可使用下列方法停用數位簽章

1. 按下 win + R 鍵 開啟 CMD 輸入視窗
2. 輸入 shutdown.exe /r /o /f /t 00
3. 在藍色畫面下選擇 疑難排解>進階選項>啟動設定>再按下“重新開機”按鈕\
4. 重新開機之後再藍色畫面按下數字鍵 7 即可停用驅動程式數位強制簽章
5. 此方法只能在該次開機停用數位強制簽章 重開機之後則自動恢復數位強制簽章

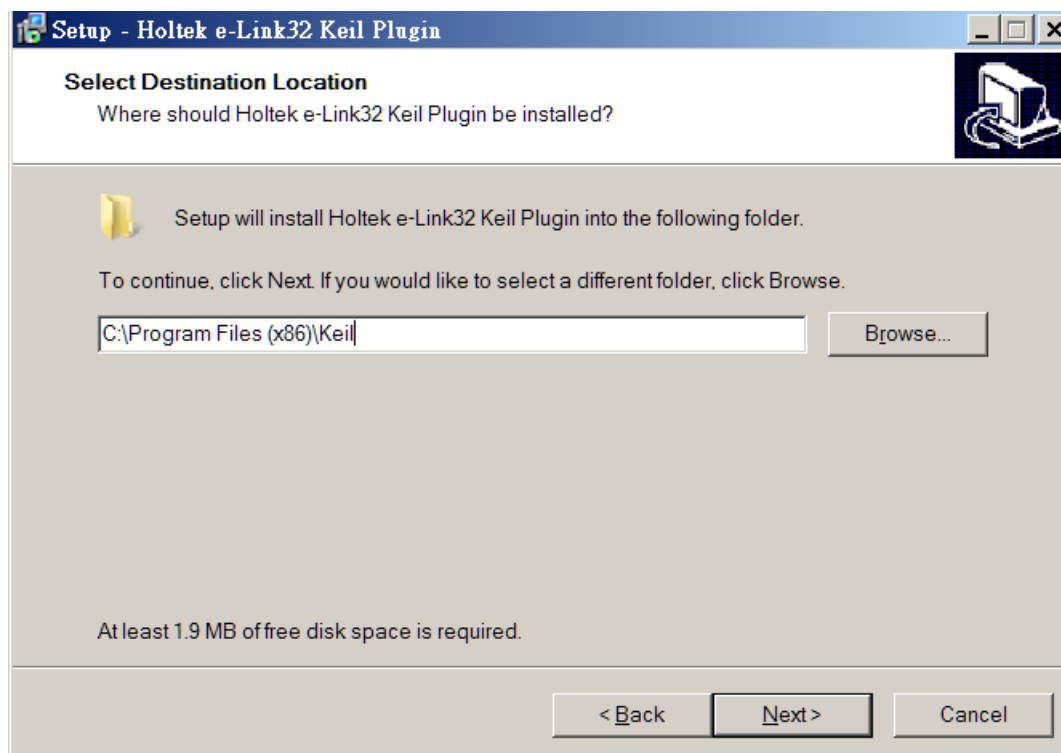
安裝 e-Link32 Keil Plugin

◦ 安裝時請以 Windows 的管理者(Administrator)權限來進行安裝。請雙點擊 e-Link32_Keil_Plugin_vXXX.exe(請至 www.holtek.com.tw 下載)圖示後開始執行安裝。

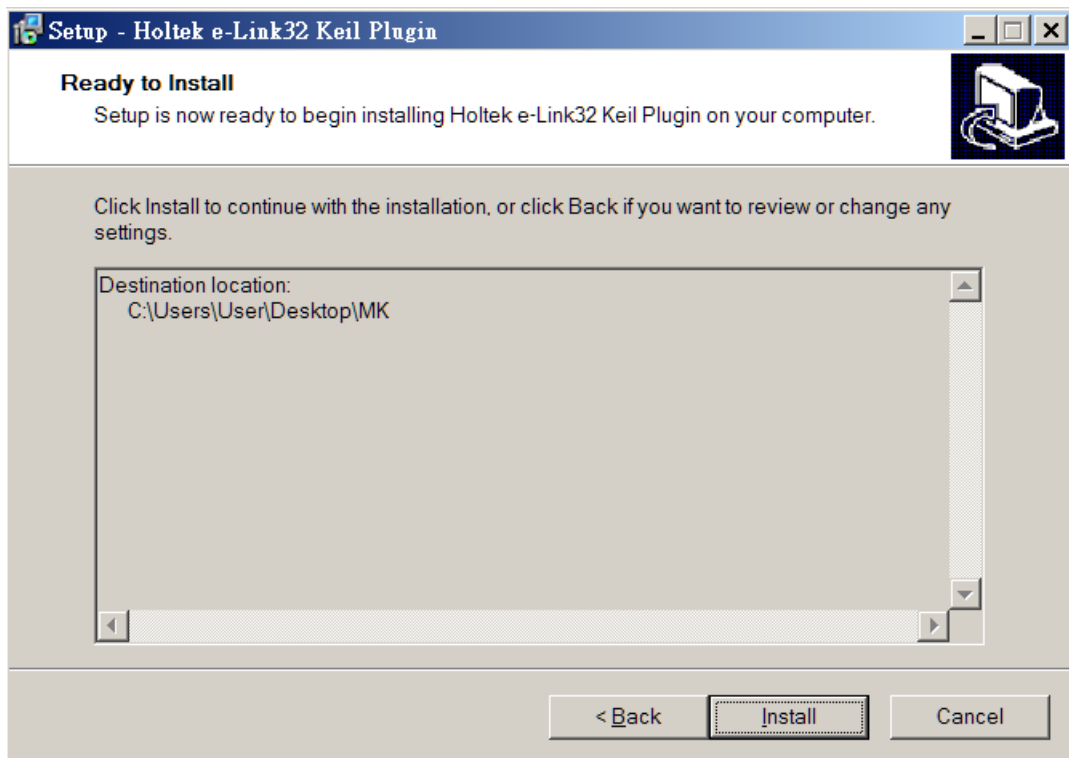
按下一步



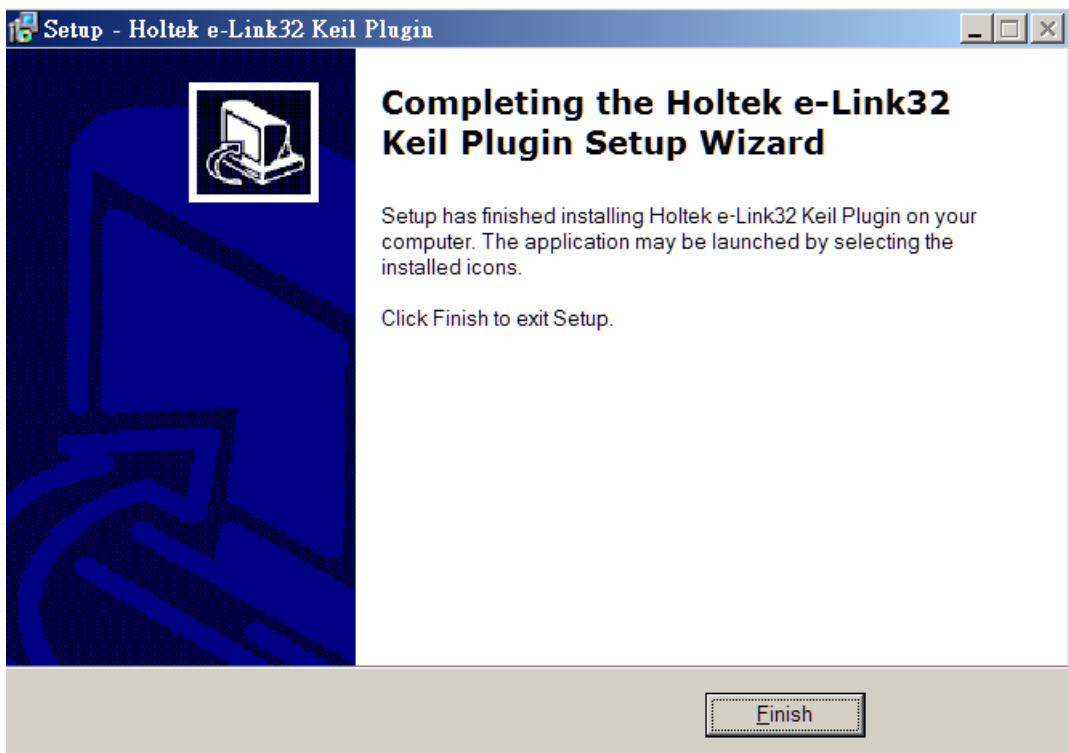
請選擇安裝至之前 KEIL 所安裝的目錄, 按下一步(Next)



選擇安裝(Install)



按下完成(Finish)即完成安裝



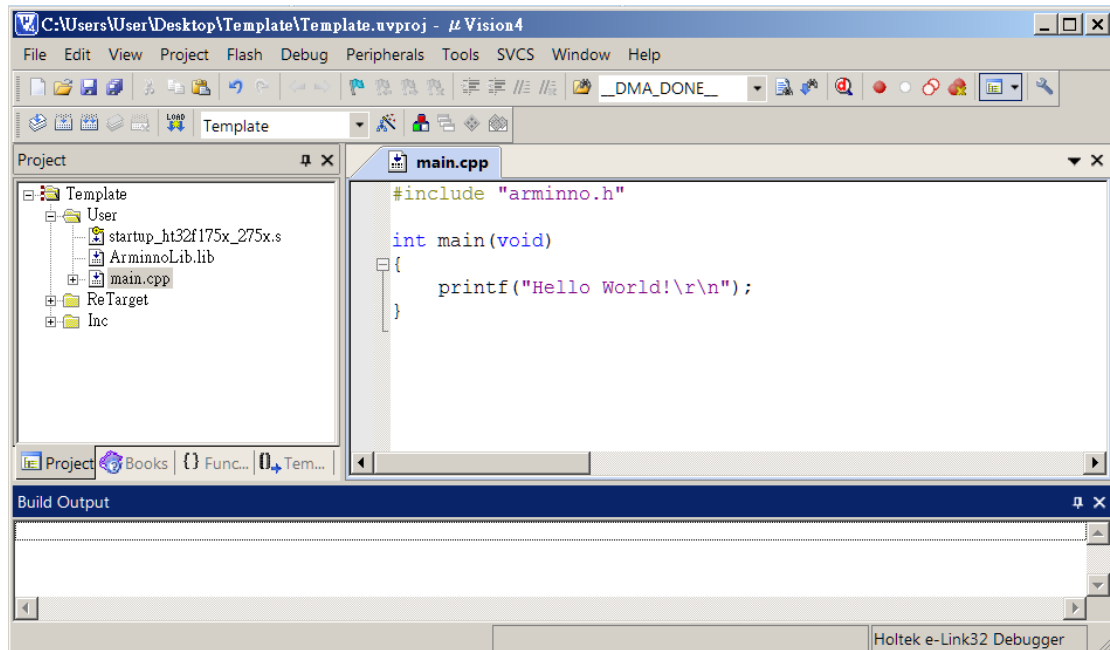
如何編寫第一個程式

為了加快開發速度，建議您使用利基所提供的 Arminno™專案開發模板(Template)。請至利基網站下載 [ARMINNO 開發環境範本](#)，並將此檔案解壓縮到您的硬碟中。目錄內容包含：

1	\Project	
2	\inc	include 檔目錄(*.h)
2	\src	程式檔目錄(*.c, *.cpp, *.lib)
2	\library	Holtek 程式庫目錄
3	\CMSIS	Cortex Microcontroller Software Interface Standard
4	\CM3	Cortex M3
5	\CoreSupport	
5	\DeviceSupport	
6	\Holtek	
7	\HT32F175x	
8	\startup	起始檔
9	\arm	Keil 起始檔目錄
9	\gcc	gcc 起始檔目錄
9	\iar	IAR 起始檔目錄
4	\HT32F175x_Driver	
5	\inc	Holtek include 檔目錄
5	\src	Holtek 程式檔目錄

開啟(Open)專案

在 Template\中找到專案檔 Template.uvproj，雙擊它後 µVision IDE 便會自動執行，開啟後畫面如下：



專案共包含三個檔案：

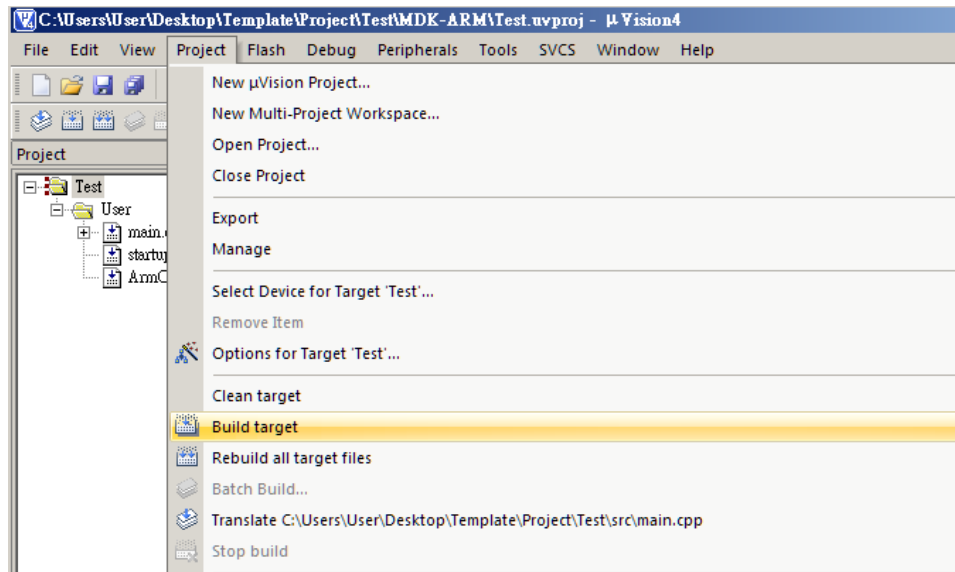
startup_ht32f175x.s 起始檔案(Startup File) 的功能是執行初始化 C 語言變數區，設定堆疊指標，中斷向量及 CPU 工作模式...等工作，等做完這些工作才會跳至使用者寫的 int main(void) 中。因此必須將 Startup File 包含進來，C 語言程式才能正常工作。您只需將 startup_ht32f175x.s 包含進您的專案中即可，不需修改它。

main.cpp 主程式 - 利基程式庫使用 C++ 開發，因此程式的副檔名必須是 .cpp。

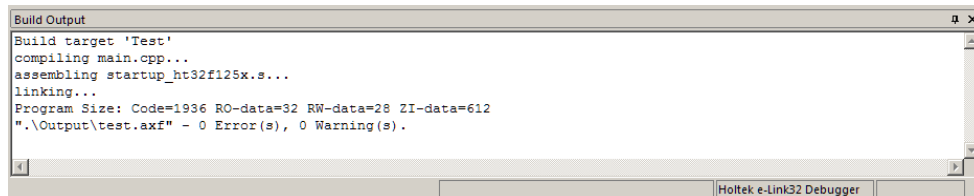
Arminno.lib Arminno™ Library 利基程式資料庫。如果要使用利基的週邊模組，或是使用利基所提供的程式庫，必須將此 .lib 包含至專案中，同時 main.cpp 也必須將 arminno.h 包含進來。

建置(Build)專案

在功能表中選擇 Project – Build target 以建置(Build)您的專案。



Build 結果顯示成功訊息。



開啟偵錯模式

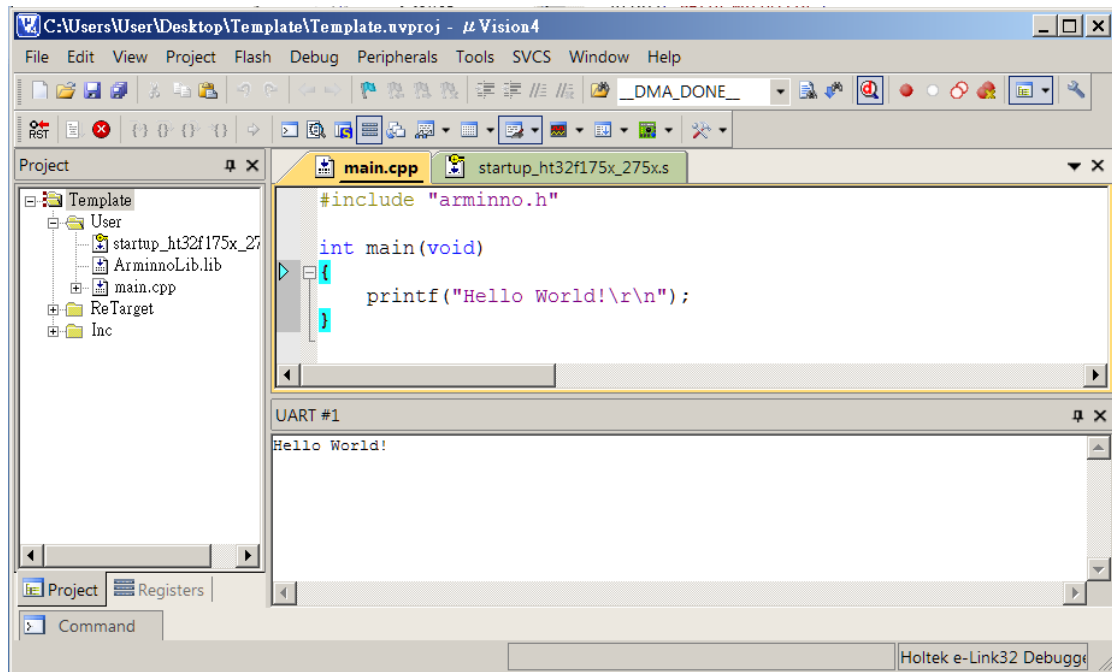
先確定：

e-Link32 驅動程式已安裝完成，且正確連接至您的電腦(USB)

在功能表中選擇 Debug –Stop/Start debug session 開啟偵錯模式(或按 Ctrl+F5)，系統會同時下載程式至開發板中

執行程式

進入偵錯模式後，在功能表中選擇 Debug – Run(或按 F5)



UART#1 視窗就可以看到程式印出 Hello World! 的結果

剛剛完成了些什麼？

我們學會了安裝 Keil μ Vision IDE，其中包含了 Real View Compiler 的整合開發環境，以及 Holtek e-Link32 驅動程式軟體的安裝。並利用利基所提供的專案開發模板，簡單編寫、建置及執行您的第一個程式

Keil μ Vision IDE 整合開發環境

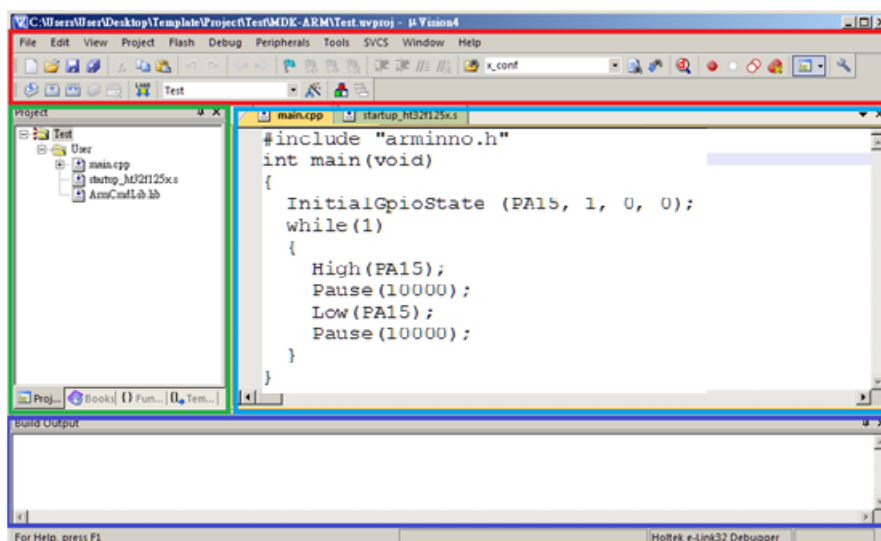
簡介

Keil μ Vision IDE 是 Keil™ 公司所開發的整合開發環境，提供 C / C++ 程式的編輯、編譯、下載、除錯等功能。

視窗各部解說

一般模式

用來編輯、建置及下載程式



功能表區 放置各個指令群組，提供使用者快速下達所要執行的指令

專案區 將開發的 .CPP 或 .S 檔加入此專案中，或從專案中刪除不必要的檔案

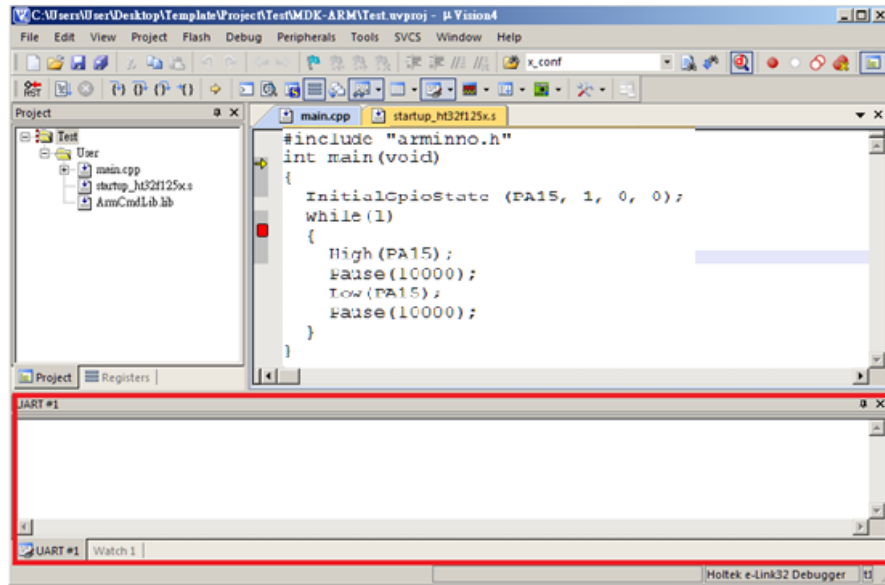
程式區 編寫程式的區域

建置輸出結果區 此區用來顯示建置(Build)程式的結果

除錯模式

當您的 eLink32 已正常連接，且提供開發板電源時，在功能表上選擇 Debug –

Start/Stop Debug Session(或按下圖示 )，就可以進入(或關閉)除錯模式



UART#1 使用 printf/scanf 的輸出入視窗。如未出現在工作視窗中，可以在除錯模式下，在功能表上選擇 View - Serial Window – UART#1(或在圖示  上選擇 UART#1)。

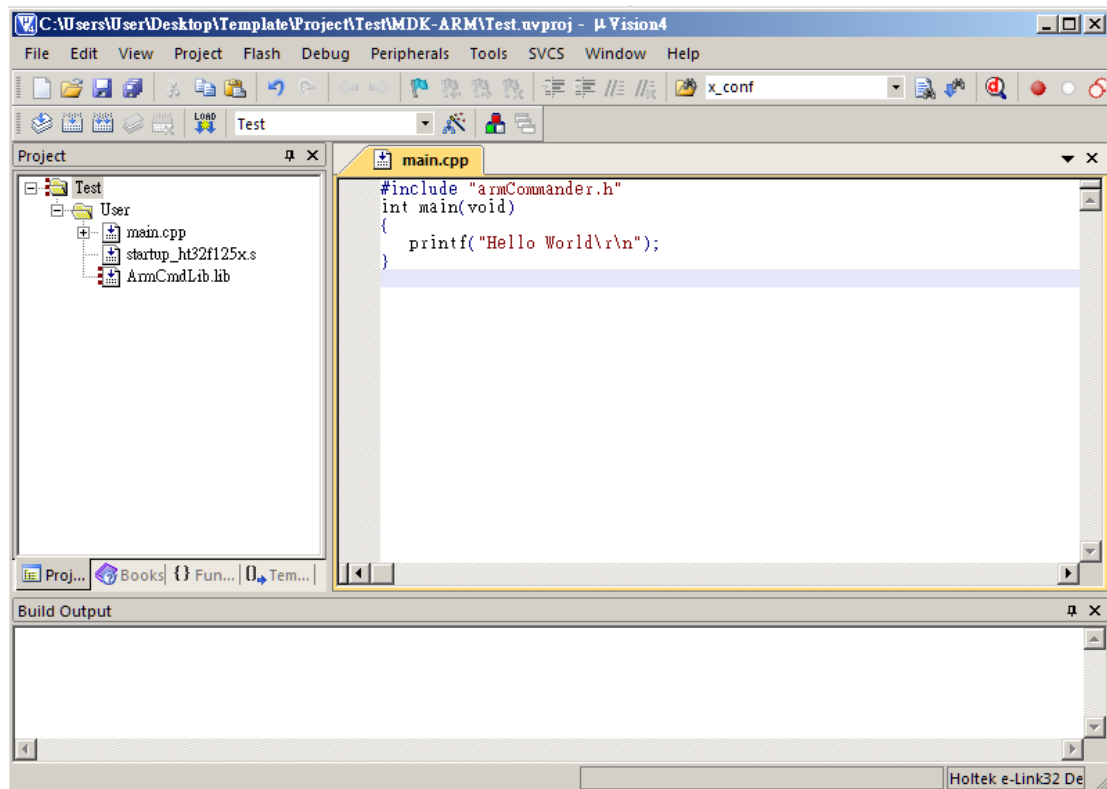
Watch 1(Watch 2) 變數監視視窗-用來監視使用者使用的全域變數。如未出現在工作視窗中，可以在除錯模式下，在功能表上選擇 View - Watch Window – Watch 1/Watch 2(或在圖示  上選擇 Watch 1/Watch 2)。

Memory 1(Memory 2/ Memory 3/ Memory 4) 記憶體監視視窗-用來監視程式記憶體。如未出現在工作視窗中，可以在除錯模式下，在功能表上選擇 View - Memory Window – Memory 1/Memory 2/ Memory 3/ Memory 4(或在圖示  上選擇 Memory 1/Memory 2/ Memory 3/ Memory 4)。

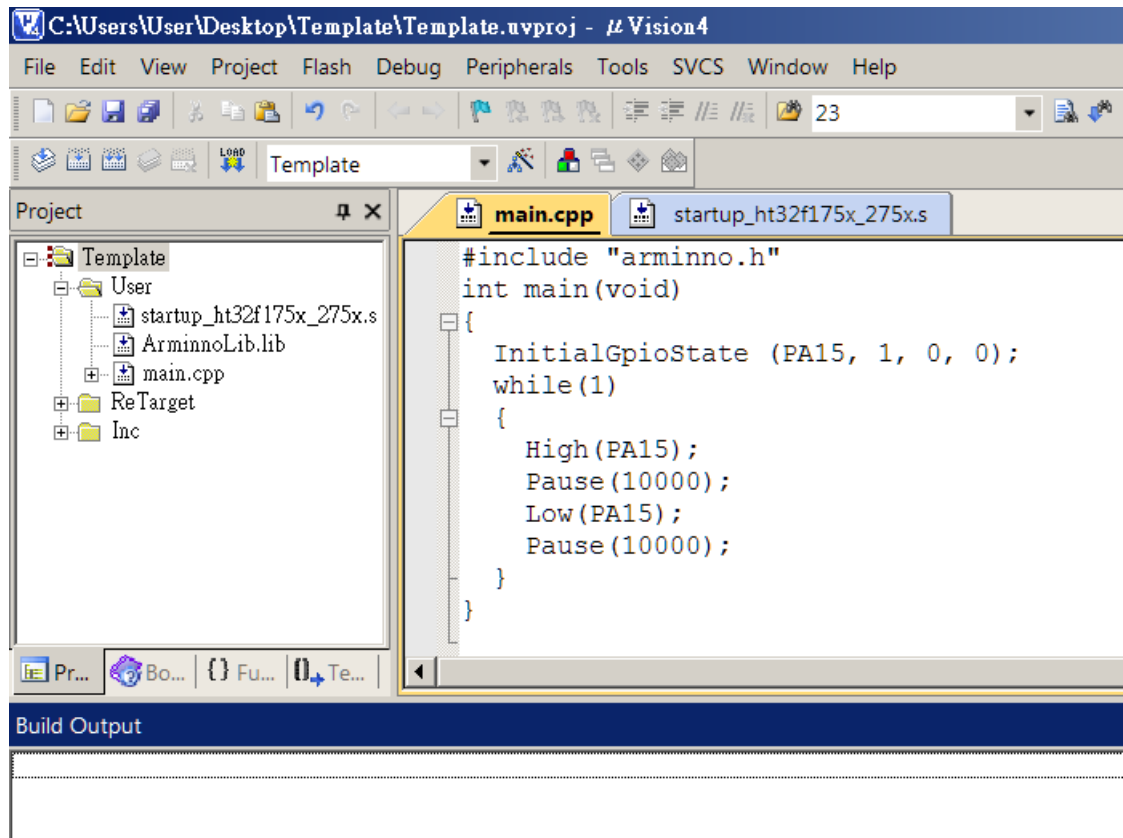
程式開發步驟

請先 ARMINNO 開發環境範本 中的 Template 解壓縮並複製到您的硬碟工作區中。

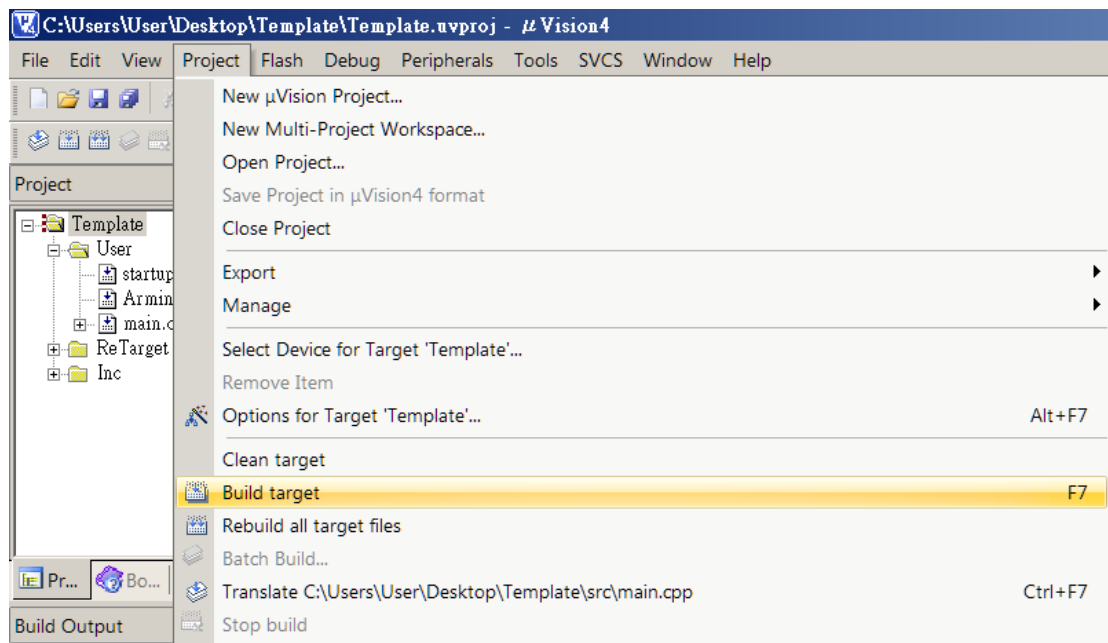
在 Template\Project\Test\MDK-ARM\中找到專案檔 Test.uvproj，雙擊它後，Keil μVision IDE 便會自動執行，開啟後畫面如下



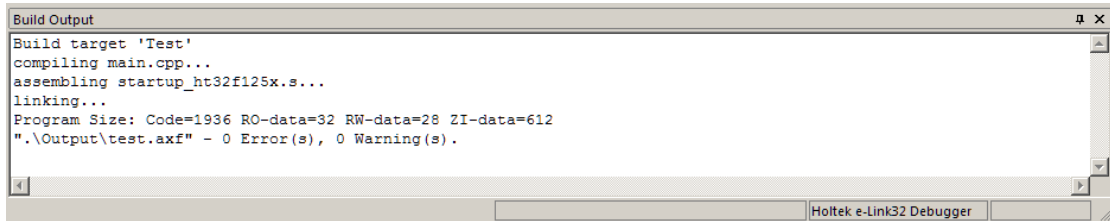
```
InitialGpioState (PA15, 1, 0, 0);
while(1)
{
    High(PA15);
    Pause(10000);
    Low(PA15);
    Pause(10000);
}
```



在功能表中選擇 Project – Build target 建置(Build)您的專案



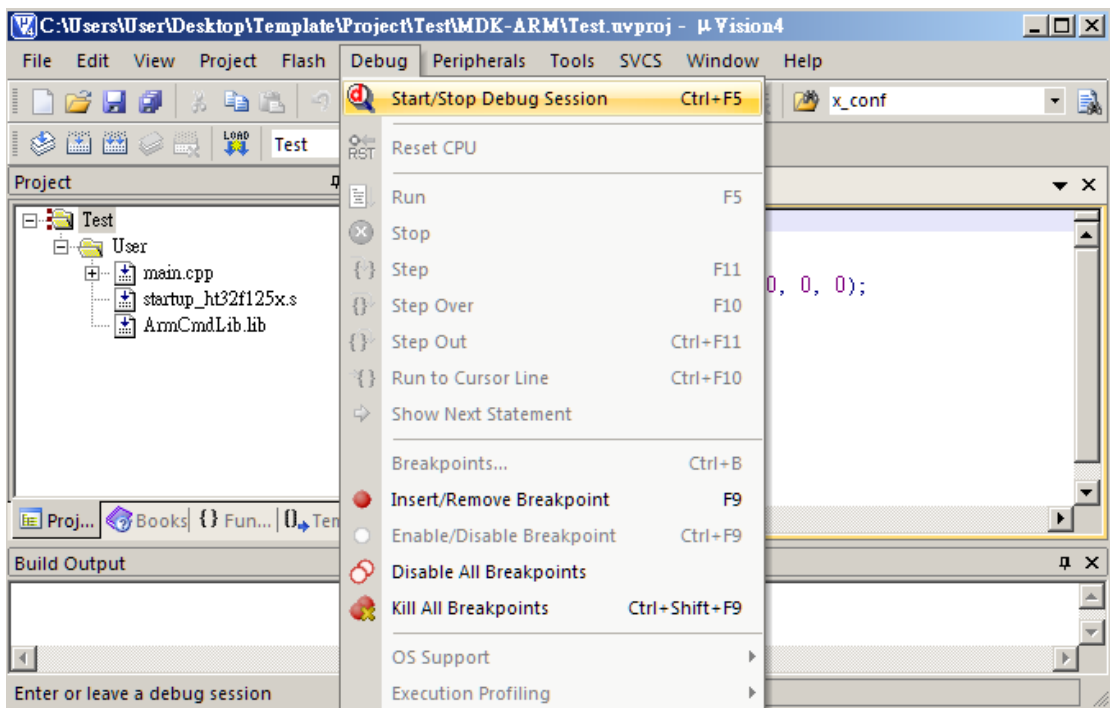
Build 結果顯示成功



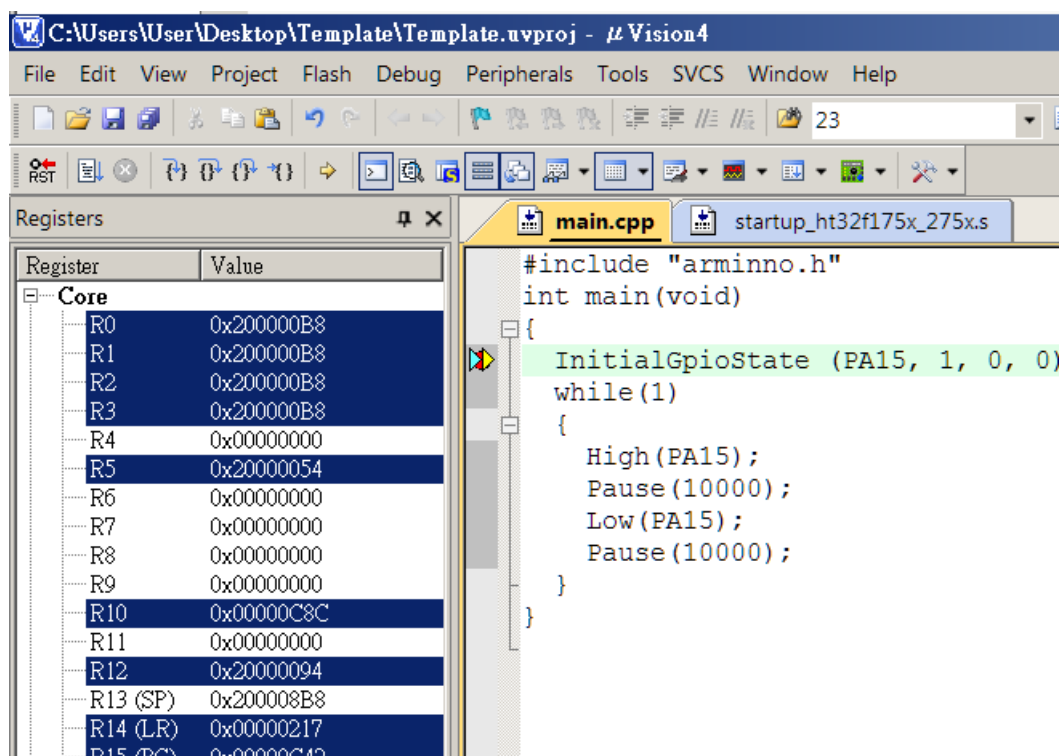
在將程式下載至 Arminno™ 開發板的前請先確定...

- 使用 USB Cable 將 Arminno 的 e-Link32 端的 USB 接口連接至 PC

選擇功能表的 Debug – Start/Stop Debug Session(或按 Ctrl+F5), 啟動 Debug 功能

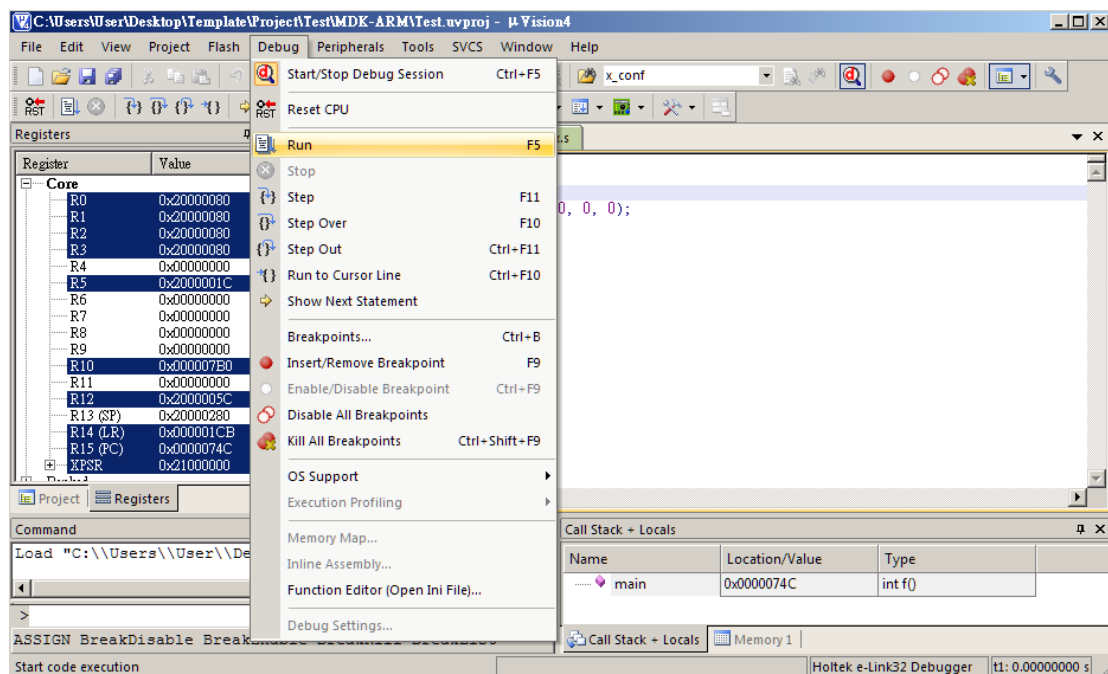


程式就會自動下載至您的 Arminno™ 開發板中



執行下載的程式

選擇功能表中的 Debug – Run(或按 F5)



觀察開發板中的 LED 是否每兩秒閃爍一次。

[注意] 下載完成後，您也可以離線執行您的程式。程式中如有呼叫 printf/scanf 等偵錯函數，在離線執行時可能會造成程式執行結果不如預期。因此執行前先請移除相關函數。

選單及命令

以下介紹最常用的選單及命令

- Project - Open Project：開啟已存在的專案。
- Project - Close Project：關閉專案。
- Project - Build target：建置專案，執行此功能將會編譯專案中的 .CPP，組譯 .S 以產生目的檔(.o)，並連結所產生的目的檔(.o)及程式庫(.LIB)產生.AXF 檔供下載至開發板中。
- Debug – Start/Stop Debug Session：開始(或結束) Debug 功能。
- Debug – Reset：重置程式。(註 1)
- Debug – Run：執行所下載的程式(註 1)
- Debug – Stop：停止所下載的程式。(註 1)
- Debug – Insert/Remove Breakpoint：插入或移除中斷點。(註 1)
- Flash – Download :將程式直接下載至開發板中。如果不要使用 ICE Debug 功能，可以選擇此功能將程式直接下載至開發板中，就可以立即離線測試。(註 2)

註 1：必須 Start Debug Session 後才可執行此功能。(啟動除錯功能)

註 2：必須 Stop Debug Session 後才可執行此功能。(關閉除錯功能)

硬體說明

簡介

本系統硬體中以 Arminno™ 最為重要，它就是一般所謂的單板電腦(Single Board Computer)，簡稱 SBC，它是整個系統的核心，也是你的專案的大腦。也可以在上面插上一些電子元件配合你的程式來完成你的專案。另外還有一些週邊硬體模組，這些模組各有各的功能，但都可以和 Arminno™ 結合成所需的專案。

Arminno™ 單板電腦

Arminno™ 是一個完整的單板電腦，它是控制整個專案的核心。它上面的主要元件包括有一顆由 HOLTEK 盛群半導體所生產之 ARM 系列工業等級微控制器 HT1755、一些時脈電路、IDE 介面及便於插件之麵包板等等，它的外觀如附圖所示。



Arminno™ 系統主要功能介紹：

■ 系統核心單元

- 32位元ARM Cortex™-M3處理器核心
- 高達72MHz工作
- 1.25DMIPS/MHz (Dhrystone 2.1)
- 單週期乘法及硬體除法

■ 記憶體單元

- 128KB Flash 記憶體用作指令/ 資料存儲

- 32KB SRAM 記憶體用作動態資料儲存

■ 時間控制單元

- 可選擇外部8MHz主振盪晶體
- 可選擇外部32,768Hz副振盪晶體
- 內部8MHz RC振盪器
- 內部32kHz RC振盪器

■ 類比數位轉換單元(ADC)

- 12-bit解析度
- 1Mps資料轉換速度
- 8個外部獨立類比輸入通道
- 轉換電壓範圍: 0V~3.3V

■ 類比運算放大器及比較器

- 2個運算放大器或2個比較器(由軟體設定配置)
- 比較電壓範圍: 0V~3.3V

■ I/O功能

- 80個通用輸入/出腳位(GPIOs)
- Port A到Port E共有16個IO外部輸入中斷(EXTI)可供使用
- 除了與類比輸入共用的腳位外(3.3V)，幾乎所有的I/O 腳位都是5V容許

■ PWM產生及捕捉定時器(Timer)

- 3個16位通用計時器(GPTM)
- 每個GPTM有4通道PWM比較輸出和輸入捕捉
- 可外部觸發輸入

■ 基本功能定時器(Timer)

- 2個32位基本計時器(BTM)

■ 真實時鐘(RTC)

- 帶可程式設計預分頻器的32 位向上計數器
- 可產生中斷溢位事件(overflow event)

■ I2C介面

- 支援100Kbps(Normal mode)和400Kbps(Fast mode)工作速度
- 支援Master or Slave模式
- 支援 7bit and 10bit Address 模式
- 支援 General Call 模式

■ SPI介面

- 最高可達18MHz工作速度
- 支援Master or Slave模式

- 8bytes資料暫存區(FIFO Buffer)

■ UART介面

- 最高可達1Mbps工作速度
- 16bytes資料暫存區(FIFO Buffer)
- 全雙工(Full Duplex, 2 wires)及半雙工(Half duplex, 1wire)支援

■ 支援除錯功能(Debug support)

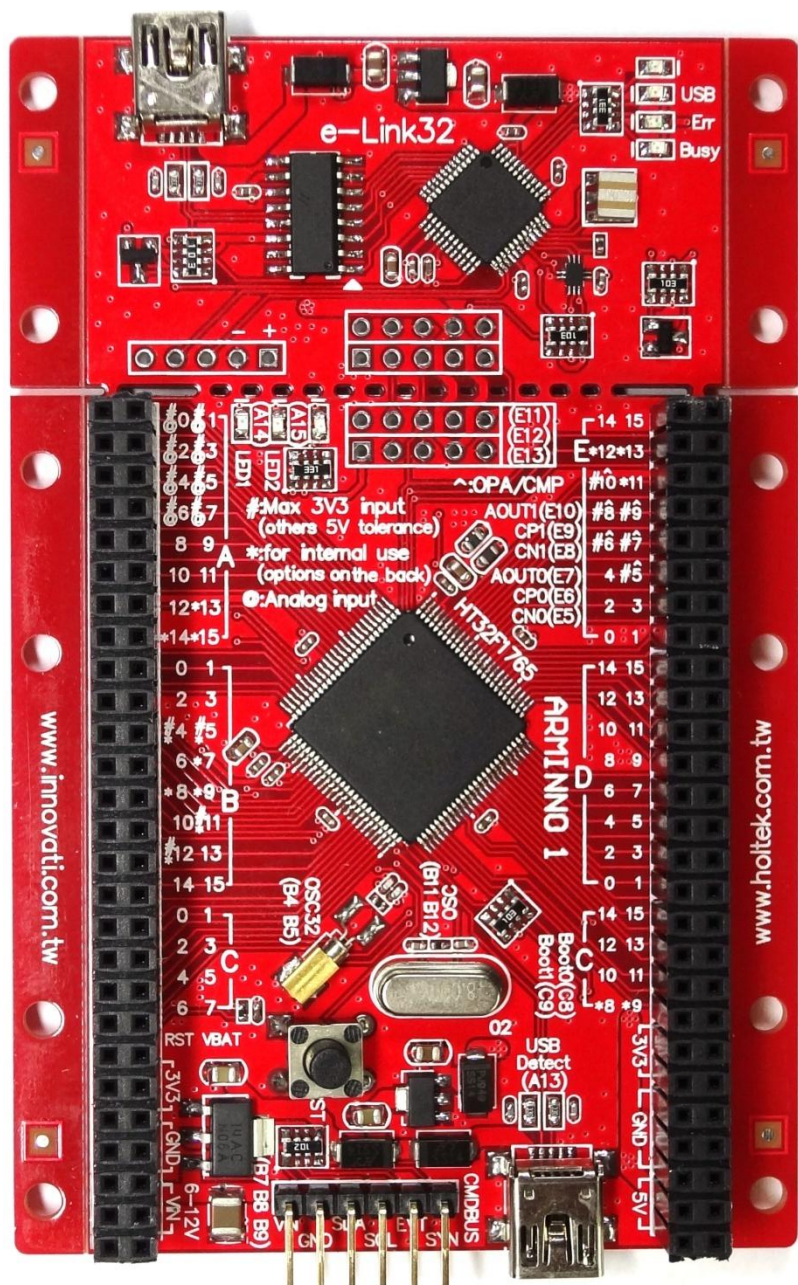
- Serial Wire Debug Port - SW-DP

■ 工作溫度: -40°C to +85°C

■ 工作輸入電壓: 6V~12VDC

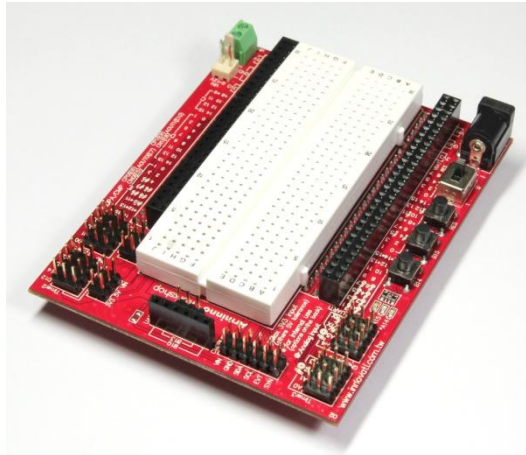
Arminno™ 應用腳位說明

Arminno™ 提供 32 組 IO 應用腳位可供使用，這 32 組 I/O 除了可以提供基本 GPIO 功能之外，Arminno™ 所提供之內建函式庫功能，經過切換之後也能藉由這 32 組某部份特定 I/O 進行輸出入控制。此外 Arminno™ 還提供常用元件及腳位接頭，如內建 LED 元件、內建外部振盪器(Resonator)、IDE 開發接頭(USB)、及 CMDBUS™ 連接座等，並透過相關跳線接頭(Jumper)設定連線與否，可讓使用者可以方便連接常用元件及功能。Arminno™ 相關應用腳位如下圖示：



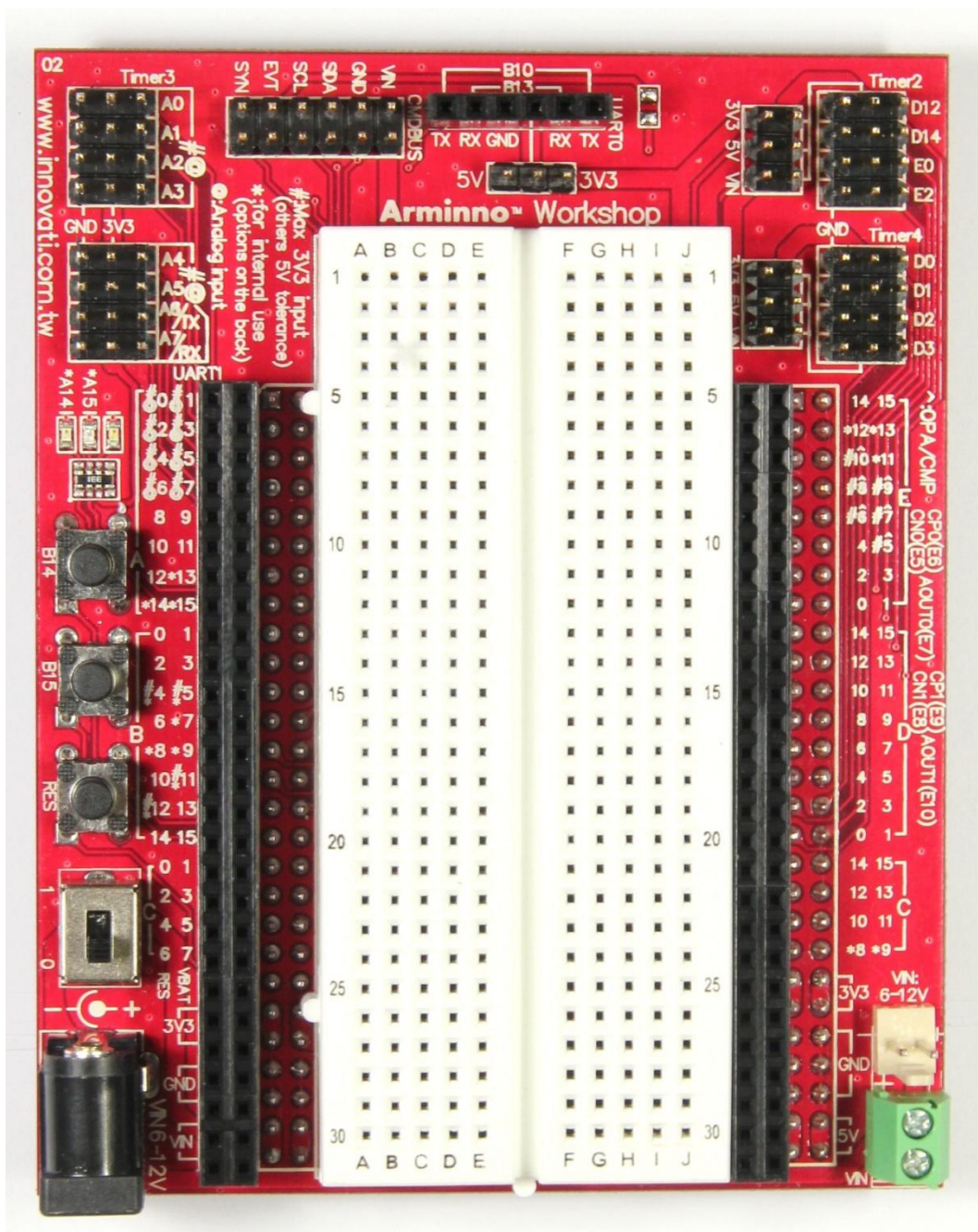
Arminno Workshop 板應用腳位說明

大部分的專案都需要外加一些零件。Arminno workshop 板上有一塊小麵包板可供使用者在上面插上所需的零件，例如：開關、發光二極體、電阻、電容等等；Arminno™板上也提供 I/O 腳的插槽，方便直接使用 Arminno™內建的 I/O 功能



Arminno™的電力來源為外加電源 VIN，使用者可選擇直接以 6-12V DC 電源插上板子上的電源插銷，或者插上一個 6-12V DC 電源供應器。Arminno workshop 板會將 VIN 穩壓成 5 伏特的 VDD 使用，其可提供的最大電流為 1 安培。此外，Arminno workshop 板有提供 12 組額外 Timer I/O 及 8 組類比輸入接頭。部份 Timer I/O 所搭配之電源接腳可於相關之跳線腳位做切換選擇，利用跳接插銷來選擇 3.3V、5V 或 VIN。

RESET 按鍵用來重置 Arminno™，程式重新開始執行。Arminno workshop 板上有兩組 CMDBUS™的接頭，可分別接上兩條 6 線的 CMDBUS™排線，用來連接利基的周邊模組。



CMDBUS™週邊模組

CMDBUS™智慧週邊模組的使用是利基 Arminno™系統的特色之一。CMDBUS™匯流排為利基針對所有 CMDBUS™模組所開發之通用匯流排，可利用定義不同 Module ID 將所有模組串接於 CMDBUS™匯流排上。



溫濕度感測模組



超音波測距模組



LCD 顯示模組



按鍵輸入模組



馬達驅動模組

CMDBUS™模組包含有 I/O 擴充模組、液晶顯示模組、馬達趨動模組、電子羅盤定位模組...等等。使用周邊模組首先須先設定模組電路板上的指撥開關，作為這模組的識別碼 (Module ID)。識別碼的範圍可從 0 到 31，請注意同一個 Arminno™控制的 CMDBUS™不能有二個模組設定相同的識別碼。每一個周邊模組接上一條 6 線的 CMDBUS™排線，再接上教育板的模組接頭。



LCD 顯示模組

背面

VIN 是未穩壓的外部電源，電壓範圍為 6~12 伏特直流電，這個電源也被穩壓成 5 伏特後供給周邊模組使用。特別注意插上 CMDBUS™時的方向，若插錯方向將導致元件損壞。當程式中宣告模組的識別碼與硬體指撥開關的設定碼相同時，這模組就可開始使用。當然，

每個模組都有它專屬的命令和功能，請參閱各模組的使用手冊。

靜電預防措施

為了方便運輸及儲存，Arminno™已經使用防靜電的包裝。但若以手拿 Arminno™時，請小心提防靜電破壞其中的 IC，這些靜電可能因為乾燥的空氣或某些特殊材料的環境造成。

C++ 程式語言

簡介

利基的 Arminno™系統使用 C++ 高階程式語言。C++是當今廣泛使用用於系統開發程式語言之一。它具有效率高、易於維護、可攜性及高度結構化之優點。其強大功能深受有經驗的使用者喜愛，同時也適合初學者學習使用。以下的例子將引導您如何使用 Arminno™，進入有趣的微控制器世界。

Hello World 程式

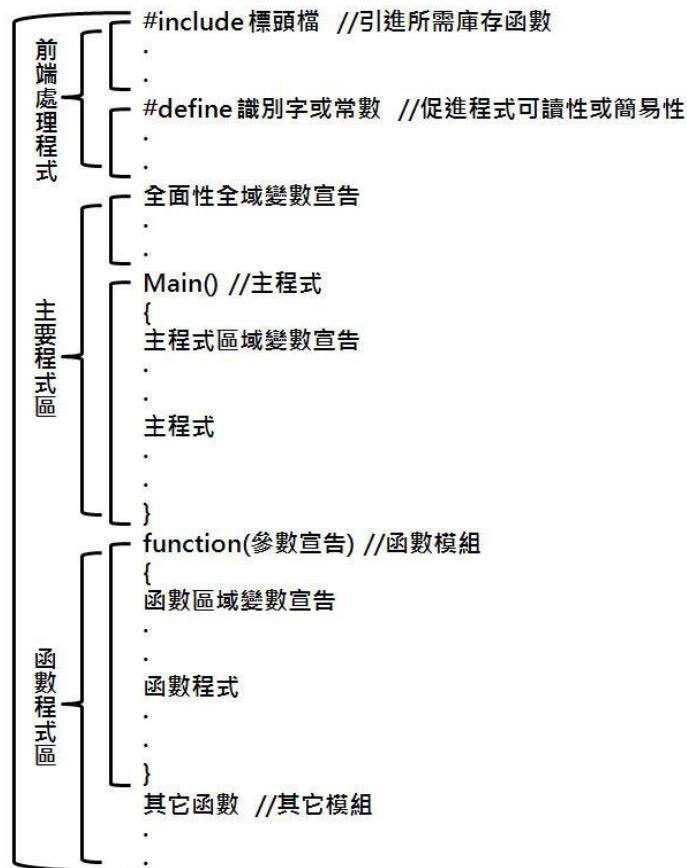
C++語言的函式是由返回值、函式名、參數列（或 void 表示沒有返回值）和函式體幾個部份所組成。下面是一個在標準輸出裝置（stdout）上，印出 "Hello World" 字串的簡單程式，通常初學程式語言時的碰到的第一個程式，就是這樣子的程式。

```
int main(void)
{
    printf( "Hello world!\n" );
    return 0;
}
```

在 C++語言中，程式會從 main 這個函式開始執行。在 main 函式中，程式設計師除了自己寫函式，也可以呼叫使用函式庫已有的函式，以進行各自項工作。例如上面程式中的 printf 函式。如果 main 是被外部其他程序呼叫，在上面的 return 0，會讓 main 返回一個數值 0 給呼叫它的程式，表明 main 程式已經成功執行。

架構及敘述 (Statements)

以下就是一個完整的程式架構圖，說明了各個組成部分的結構與順序。第一區為前置處理區，用來引進所需的函式庫，以及定義識別字增加程式閱讀性或程式編譯彈性；第二區為主要程式區，前面可以宣告全域變數，然後則是 main 這個函式。在 main 這個函式內則包含了可以宣告 main 函式的區域變數，以及程式。前面有提到，程式會從 main 這個函式開始執行；第三區則為函式區。所有使用者自己開發的函式，就都放在這一區。各個函式內可以宣告該函式的區域變數，以及程式。



編譯器會將程式中的敘述編譯成可執行的對應程式碼。敘述可以包含常數、變數、運算式以及函數，用來定義常數、宣告變數、執行算術及邏輯運算、以及執行程式控制轉換及宣告函式。

註釋 (Comments)

以 `//` 開頭的文字用來代表註釋，例如：

```
int i=3; //Here is the comment!
```

在 `//` 右邊的文字都被視為註釋，不會被編譯器所編譯。請注意`//`只對單一行有效，註釋不能跨行連接。

識別字 (Identifiers)

識別字是 C++ 語言系統上已內建好的一些型態識別字、函數名稱等，屬於 C++ 語言

上所能提供之功能名稱，因此使用者不可以用來當做自訂變數及自訂函數名稱。識別字是使用者給變數、函數等命名的名稱。關鍵字及內建識別字不能當作識別字使用，識別字必須以英文字或底線符號開頭，後面跟著英文字母、數字、底線或金錢 (\$) 符號。識別字最大長度為 32 個字元，英文字母大小寫，代表不同的識別字。

關鍵字 (Keywords)

關鍵字是一些有特殊意義的保留字，不可以被拿來當識別字使用。以下為部份關鍵字：

auto	double	int	struct
break	else	long	Switch
case	enum	register	typedef
char	extern	return	union
const	float	short	unsigned
continue	for	signed	void
default	goto	sizeof	volatile
do	if	static	while

標籤 (Labels)

標籤必須出現在行的最前面，而且必須是合法的識別字，後面以冒號結束，它用來標示一段可供 GOTO 敘述跳躍的程式開頭。

常數、變數及資料型態 (Constants, Variables and Data Types)

常數是一個固定值，它不會因為程式的執行而改變，宣告時須以關鍵字 `const` 宣告，其型式如下，例如：

```
const Type Constname = expression  
或  
#define Constname expression
```

以下是分別以位元組、字串、及陣列宣告常數的例子：

```
const int DaysofMay =31;
const char Name[7]= "January" ;
const unsigned char Number[4]={23,34,45,61};
```

常數值需符合所給的型態。常數有可被整個程式取用的意義，所以它必須以全域宣告。若宣告一個常數陣列時，其元素不可為常數字串，或另一個常數陣列。相對於常數而言，變數所代表的值則可能在程式執行過程中被改變。宣告方式如下，例如：

```
Type Variablename [= value];
```

任何變數在被程式使用前，必須先被宣告，以及它的型態、大小等。變數須先設定初始值，系統預設的初始值是 0 或空的字串。慣例上，變數的原始資料型態為 char、unsigned char、short、unsigned short、int、unsigned int、float、double、long long、unsigned long long 等，且均會實際佔用記憶體的字址。如果被宣告在程式裡，那麼這變數就屬於區域性變數，只能給所在的程式區塊使用；如果是宣告在程式外面，那麼就屬於全域變數，就可以讓全部的程式使用。

型態轉換

資料型態的轉變是指轉換變數儲存的資料，而不是變數本身。轉變型態通則是從較窄的運算域轉到較寬的運算域，例如從 Short 轉成 Int 或 long 型態。如果反過來操作，那麼多出來的高位元資料將會遺失，須小心使用。

下表為 C++ 程式語言所支援所有變數型態。

變數型態	大小	內容
char	1 Byte	有符號變數，值域為 -128~+127
unsigned char	1 Byte	無符號變數，值域為 0~255。
short	2 Byte	有符號變數，值域為 -32768~+32767
unsigned short	2 Byte	無符號變數，值域為 0~65535
int	4 Byte	有符號變數，值域為 -2147483648~2147483647
unsigned int	4 Bytes	無符號變數，值域為 0~4294967295
float	4 Bytes	浮點數變數，值域為 3.4E-38~3.4E+38。
double	8 Bytes	浮點數變數，值域為 1.7E-308~1.7E+308。
long	4 Bytes	有符號變數，值域為 -2147483648~2147483647

unsigned long	4 Bytes	無符號變數，值域為 0~4294967295
long long	8 Bytes	有符號變數，值域為 -9223372036854775808 ~ 9223372036854775807
unsigned long long	8 Bytes	無符號變數，值域為 0~18446744073709551616

整數值 (Integral Literals)

整數文字值可以為十進位、十六進位。十進位元文字值是一串不需要字首的十進位數字(0-9);十六進位元文字值是一串以 **0x** 為字首的十六進位數字(0~9, A~F);十進位元文字值直接表現出整數文字值的十進位元值。

浮點數值 (Floating-Point Literals)

浮點數值包含一個整數、一個可有可無的十進位點(ASCII 碼的句點)及小數、以及一個可有可無的以 10 為底的指數。

陣列 (Arrays)

陣列裡含有一些元素。擷取這些元素都必須以索引的方式。如果一個陣列有二組索引，則稱為二維陣列，以上則以此類推。陣列宣告時可以在名稱後面順便宣告陣列的大小。陣列的索引是從 0 開始計算。例如：

```
int myvector [10];      // 從 myvector[0]至 myvector[9],共 10 個元素
char mystring [ 50];    // 從 mystring[0]至 mystring[49],共 50 個元素
int mymatrix [3][3];    // 從 mymatrix[0][0]至 mymatrix[2][2],共 9 個元素
```

以下例子為利用陣列寫出九九乘法表。

```
int main(void)
{
    unsigned char m[9][9];
    unsigned char i,j;
    for(i=0;i<=8;i++)
        for(j=0;j<=8;j++)
            m[i][j]=(i+1)*(j+1);
```

```
}
```

完成程式之後進入 debug 模式，將程式執行到最後，並透過 watch window 來觀看 m 陣列，裡頭即會有完整九九乘法表。

指標

如果一個變數宣告時在前面使用 * 號，表明這是個指標型變數。換句話說，該變數儲存一個位址，而 *則是取得該位址所指向的記憶體內容的一元運算子。指標不僅可以是變數的位址，還可以是陣列、函式的位址。透過指標作為 return 形式參數可以在函式呼叫過後，取得到多個返回值。

透過指標有些情況下可以執行得更為有效率，但是不正確的或者過度的使用指標又會給程式帶來許多潛在的錯誤機會，使用指標時要留意。以下就是使用指標的範例。例如：

```
char *mychar;    // 指向字元資料的指標變數
int *ap[3];      // 指向長度為 3 的整數型資料陣列的指標變數
int **argv;      // 指向一個整數資料的指標的指標變數
```

另一個運算符 &，叫做取位址運算符，它將返回一個變數、陣列或函式的儲存位址。因此，下面的例子：

```
int i, *pi;    // 整數變數 i 與指標變數 pi
pi = &i;      // i 的位址儲存到指標變數 pi
```

只要 pi 指向 i 變數, i 和 *pi 在程式中可以相互替換使用的。

運算子 (Operators)

包含二種運算子：單元運算子 (Unary operators) 是以一個前置的符號操作一個運算元，例如正、負符號。雙元運算子 (Binary operators)則是以一個置中符號操作二個運算元，例如加、減號。當運算式中含有多個運算子時，運算子的優先序如下表，當然，你可以使用括弧來改變優先順序。

運算子優先順序由高而低排列如下：

類別	運算子
遞增 1, 遞減 1	++, --

正號、負號、邏輯非和邏輯反	+, -, !, ~
乘、除及餘數除法	*, /, %
加和減	+, -
移位	<, >
邏輯小於、大於、小於等於及大於等於	<, >, <=, >=
邏輯等於和不等於	==, !=
位元運算 AND、OR、XOR 及補數	&, , ^ 及 ~
邏輯 AND 及 OR	&& 及

以下介紹各種不同的運算子。

算術運算子

共有七種算術運算子，

- + 加
- 減
- * 乘
- / 除
- % 餘數 (整數除法餘數)
- << 左移 (等同二進位乘法)
- >> 右移 (等同二進位除法)

關係運算子

關係運算子比較二個數值並回傳一個「真」(1) 或「偽」(0) 的結果。

- > 大於
- >= 大於等於
- < 小於
- <= 小於等於
- == 等於
- != 不等於

位元運算子

- ~ 位元 1 的補數
- & 位元 AND
- | 位元 OR
- ^ 位元 XOR

邏輯運算子

邏輯運算子支援下列邏輯運算，以產生「真」(1) 或「偽」(0)的運算結果。

&&	邏輯 AND
	邏輯 OR
!	邏輯 NOT

&&與||所連結的表示式，其運算次序由左向右，如關係表示式或與邏輯表示式為「真」時，其對應數值為 1，反之則為 0。!單元運算子會將非零數值轉為 0，而將 0 數值轉為 1。

指定運算子

指定運算子(Assignment Operators)最簡單的運用是將等號右邊的變數值取代等號左邊的變數值。其他五種是混合的指定運算子。以 $A += B$ 為例，將等同於寫成 $A = A + B$ 。

=
+=
-=
*=
/=
%=
++
--

上述最後二項則是遞加 1 或遞減 1 指定運算子，用法上分為前綴與後綴，以變數 a 為例，即有下列四種寫法。

++a	前綴遞加，變數 a 使用前先遞加 1
--a	前綴遞減，變數 a 使用前先遞減 1
a++	後綴遞加，變數 a 使用後才遞加 1
a--	後綴遞減，變數 a 使用後才遞減 1

程式控制流程

程式敘述以其在程式中的順序依序被執行，但可使用條件敘述或是跳轉敘述來改變其執行順序。條件敘述有四種：if.....else 敘述、switch...case 敘述、while..., do...while 敘述與 For...敘述。

if 敘述

兩種 if 包括：

```
if (條件式)
    敘述;

以及

if (條件式)
    敘述;
else
    敘述;
```

條件式的值非零表示條件為真；如果條件為假，程式將跳過 if 處的敘述，直接執行 if 後面的敘述。但是如果 if 後面有 else，則當條件為假時，程式跳到 else 處執行。if 和 else 後面的敘述可以是另一個 if 敘述，這種套疊式的結構，可以進行更複雜的邏輯控制。原則上 else 一定與上方最接近的 if 成對，否則須以大括弧{}指定配對關係。請比較下列兩種狀況：

```
if (條件式)
    if (條件式)
        敘述;
    else
        敘述;

if (條件式) {
    if (條件式)
        敘述;
}
else
    敘述;
```

switch...case 敘述

```
switch (運算式) {
    case 值 1:
        敘述;
        break;
    case 值 2:
        敘述;
    default:
        敘述;
```

```
}
```

switch 通常用於對幾種有明確值的條件進行控制。它要求的條件值通常是整數或字元。與 switch 搭配的條件轉移是 case。使用 case 後面的標值，控制程式將跳到滿足條件的 case 處一直往下執行，直到 switch 敘述結束或遇到 break。使用 default 把其他例外的情況包含進去。如果 switch 敘述中 case 的值均不符合時，控制程式將跳到 default 處執行；如果省略 default 子句，則離開 switch 敘述直接執行下一敘述。case 後面的敘述可以是另一個 switch 敘述，這種套疊式的結構，可以進行更複雜的條件控制。

while 敘述

```
while (條件式)
    敘述;
```

當 while 語句條件式的值為真 (非零值) 時，其下的程式段將被執行。可以使用 break 語句在程式段之中，用來立即離開 while 迴圈。

do...while 敘述

do...while 敘述可依據條件重複執行某敘述。

```
do
    敘述
while (條件式);
```

基本的 do ...while 指令會讓其中的敘述永遠不斷地執行。因為 while 被放在迴圈結束的地方，則程式段將至少被執行一次，然後才做條件式值的判斷。可以使用 break 語句 e 在程式段之中，用來立即離開 do 迴圈。

for 敘述

如果一個程式要做一些已知次數的循環，則可使用 For 敘述，配合迴圈變數的增加或減少來做循環次數控制。

For 敘述需有一個迴圈變數、一個起始值、一個結束值和一個間距值。程式在 For 之後將以起始值代入迴圈變數開始執行，以後每執行一次程式段，變數值就以後面的間距值增加或減少，直到變數值等於結束值才離開程式迴圈去執行之後的敘述。

```
for (表達式 1; 表達式 2; 表達式 3)
    敘述;
```

你可以使用關鍵字 `break` 跳出 `for` 迴圈，但不建議從迴圈外直接跳入迴圈內，這樣潛在造成迴圈控制的錯誤。

goto、continue、break 及 return 敘述

跳轉敘述包括四種：`goto`、`continue`、`break`、`return`。

`goto` 敘述是無條件轉移敘述：

`goto` 標記；

標記必須在當前函式中定義，使用「標記：」的格式定義。程式將跳到標記處繼續執行。由於 `goto` 容易產生閱讀上的困難，且程式缺乏結構性維護困難，所以應該儘量避免使用。

`continue` 敘述用在迴圈敘述中，作用是立即結束當前的迴圈，馬上開始下一輪迴圈。

`break` 敘述用在迴圈敘述或 `switch` 中，作用是立即結束當前迴圈，跳到外層迴圈繼續執行。但是使用 `break` 只能跳出一層迴圈。在要跳出多重迴圈時，可以使用 `goto` 敘述。

當執行到函式中的 `return` 敘述時，不論是否還有後續的其他敘述，皆會結束函式並返回。`return` 可以跟一個運算式或變數作為返回值。如果 `return` 後面沒有值，則無返回值。

函式 (Functions)

程式中多次重複使用的程式碼，可以考慮將之定義為函式，以便重覆呼叫使用，一來減少程式空間需求，同時結構化的函式更可以降低程式維護成本。在 C++ 中函式的組成包括四個部份，即返回值型別、函式名稱、參數(parameter)列與函式主體。當呼叫一個函式，須傳入函式內部的引數(arguments)，以及函式執行後，須傳回的運算結果(return value)，稱之為函式的傳回值。在函式被呼叫之前，必須先作宣告，否則編譯時會產生錯誤訊息。以下是一個函式的組成範例：

```
傳回值型別 函式名稱(參數型別 1 參數 1, 參數型別 2 參數 2, ...)  
{  
    宣告變數;  
    敘述;  
    return 傳回值;  
}
```

如果呼叫函式不傳回任何值，則宣告其傳回值型別為 `void`，若不需傳入任何引數，則

參數列保持空白即可。參數傳遞在 C++ 語言裡，有 3 種方法，分別是 call by value、call by pointer 與 call by reference 的方法。

Call by value

參數以數值方式傳遞，複製一個副本給另一個函式，例如：

```
int main() {
    int z = 12;
    func(z); // 傳值給函式
}

void func (int x) {
    x++; // 修改此 x, main 的 z 值不受影響
}
```

Call by pointer

Call by pointer 是將變數的位址傳到函式，而函式使用一個指標接住這個位址，因此函式的這個指標可以指向並修改這個數值。例如：

```
int main() {
    int z = 12;
    func(&z); // 加&, 傳位址給函式
}

void func (int *x) {
    *x++; // 修改此 x 就是修改 main 的 z
}
```

Call by reference

Call by reference 的功能和 Call by pointer 是一樣的，都是指到原本變數的位址所以可以直接修改其內容。但是在傳遞變數到函式時，不用加&，在函式中引用變數也不用加*號就可以直接使用該變數。但是在函式裡的參數裡加上&，表示是 call by reference，程式寫起來較為簡潔。例如：

```
int main() {
    int z = 1;
    func(z); // 不用加&
}

void func (int &x) {    // 參數前要加上&，表示是 call by reference
    x++;               // 修改此 x 就會直接修改到 main 的 z 變數
}
```

```
}
```

函式裡的參數裡加上&，代表是 Call by reference。

volatile 宣告

在 C 裡面有 volatile 這個宣告，通常是說這個變數會被外在 routine 改變，在 kernel 裡面通常是指會被 interrupt handler (有時就是硬體中斷的 routine) 改變值，也就是被非同步的改變的變數。例如 unsigned long volatile i; i 在 kernel 是時間每次 hardware 的中斷會來改這個值 在 asm 裡面是說這個東西 compiler 時，compiler 不要作 optimized，因為最佳化的結果，compiler 會把 code 按照他想的方法放到記憶體裡，但是有的 code 我們需要特定指定他一定要在某個記憶體上，在 kernel 裡常有這樣情形發生而造成輸出結果不合乎預期。

在這裡例子中，代碼將 foo 的值設置為 0。然後開始不斷地輪詢它的值直到它變成 255：

```
static int foo;
void bar(void) {
    foo = 0;
    while (foo != 255)
        ;
}
```

一個執行優化的編譯器會提示沒有代碼能修改 foo 的值，並假設它永遠都只會是 0。因此編譯器將用類似下列的無限迴圈替換函數體：

```
void bar_optimized(void) {
    foo = 0;
    while (true)
        ;
}
```

但是，foo 可能指向一個隨時都能被電腦系統其他部分修改的位址，例如一個連接到中央處理器的設備的硬體暫存器。上面的代碼永遠檢測不到這樣的修改。如果不使用 volatile 關鍵字，編譯器將假設當前程式是系統中唯一能改變這個值部分 (這是到目前為止最廣泛的一種情況)。為了阻止編譯器像上面那樣優化代碼，需要使用 volatile 關鍵字：

```
static volatile int foo;
void bar (void) {
    foo = 0;
    while (foo != 255)
        ;
}
```

這樣修改以後迴圈條件就不會被優化掉，當值改變的時候系統將會檢測到。

static 宣告

通常函式裡宣告的變數，在函式被呼叫期間變數可以保持它們的空間及數值，但是一旦結束呼叫，則相關所有在函式裡宣告之變數的記憶體空間會被釋放。當下一次執行該程序時，它的所有區域變數將重新分配而造成變數資料內容消失。

然而，您可以將區域變數宣告成靜態，以保留變數的值。在程序內部用 **Static** 關鍵字宣告一個或多個變數，使變數記憶體不因函式結束呼叫而被釋放，可以保存相關變數中之數值供之後使用。

```
static unsigned int MyDate;
```

extern "C" { ... }

C 語言和 C++ 語言的 Compiler 對於函式的編譯方式是不同的

如 `void func(int x, int y);`

該函式被 C Compiler 編譯成 `_func`，但是如果是 C++ Compiler，就會變成 `_func_int_int`。請注意，不同的 compiler 可能會產生不同的名字，可由查閱產生的 .map 檔來得知實際的函數名稱。如果都是用同一種語言來寫，程式設計者是不用理會這些差異，但是如果同一專案內混合 C 與 C++ 或組合語言，那就必須瞭解被呼叫的函數是使用何種語言撰寫，這樣才能被呼叫到。為了簡化流程，如果使用 C 語言寫的程式，其 .h 檔中的最前和最後最好加入下列語法：

```
#ifndef __cplusplus           // 寫在第一行，表示括號內的所有函式都是用 C 寫的
    extern "C" {
#endif
```

```
.....  
#ifdef __cplusplus    // 寫在最後一行  
    }  
#endif
```

所以當使用 C++ 語言去呼叫 C 語言所寫的函數, 就不會有呼叫不到的問題,(即使寫對函式名稱)而發生 Undefined symbol xxx() (referred from yyyyy.o). 的問題

利基函式庫

針對 Arminno™ 微控制器內建之硬體功能，利基提供了一系列之內建功能相關函式庫以供 User 使用，這些函式庫不僅易學易懂，更與內建相關硬體做緊密及高效率的結合，可謂兩者兼顧。

函式庫介紹

下表為函式庫分類介紹表：

名稱	說明	
GPIO	一般 I/O 輸出和輸入控制功能設定	
ADC	類比訊號轉數位資訊輸入控制	
Timer 0 & Timer 1	內建之兩組基本功能計時器，無任何外部 I/O 功能	
Timer 2、Timer 3 & Timer 4	內建之三組多功能計時器，可執行多種計時器相關功能，功能如下	
	Counter	固定時間週期觸發，及外部計數功能
	PWM	輸出脈衝寬度調變(Pulse Width Modulation)波形
	PulseIn	量測單一輸入脈波寬度
	PulseOut	輸出單一不同寬度之脈波
	FreqOut	輸出不同頻率之脈波
	Decoder	提供二線編碼器解碼功能
RTC	次要時鐘來源輸入，可像時鐘一樣輸出實際時間之計數器功能	
OPA0/CMP0 & OPA1/CMP1	內建之二組放大器和比較器	
EEDATA	內建之 EEPROM 功能	
I2C	內建兩組 I2C 通訊介面，支援 Master 和 Slave 雙模式	
SPI	內建兩組 SPI 通訊介面，支援 Master 和 Slave 雙模式	
UART	內建兩組 UART 通訊介面	
USB	內建 USB 通訊介面，可以和電腦 PC 交換傳遞資訊	
其它系統相關 函式功能	Pause	時間延遲指令
	SetMainClk	切換主要時鐘源

	SetX32Clk	切換次要 32,768Hz 時鐘源
	GlobalEventControl	開啟或關閉所有 Event
	GetLibVer	讀取 Library 版本資訊
	SetIdeOff	取消 IDE 功能並釋放 I/O 腳位
	SetClkOut	設定時鐘源輸出功能

Arminno™所提供之函式庫功能，會依需求不同而搭配相關 I/O 使用。一般而言，在初始設定之下，所有 I/O 都會設定為 GPIO 模式(除了 X32、XTAL 及 IDE 功能)(請參考：[I/O 功能腳位對應表](#))，若是啟動非 GPIO 內建函式庫功能則會佔用其所對應之相關 I/O，並取代該 I/O 之 GPIO 設定。請注意同一 I/O 有可能會有多個不同的非 GPIO 內建函式庫可以設定，若是重複設定多個非 GPIO 內建函式庫功能至相同 I/O 則會有函式庫啟動失敗的情況發生(只有最先設定之功能會成功)。要避免失敗則需先取消前一個相同 I/O 設定函式庫功能，之後再啟動下一個相同 I/O 之函式庫功能。因此使用者在使用函式庫功能之前需先妥善規劃其 I/O 分配，以達到最佳使用效果

Arminno™所提供之函式庫功能，其各個函式會根據不同需求而要求一個或多個參數來完成函式設定。然而在大部分的 Arminno™函式應用裡，使用者不需要對該函式的所有參數來做設定，而只需要對該函式裡需要的參數來做設定，並讓該函式其他的參數使用函式本身預設值輸入即可。例如下列 SetUart0()函式。

```
State = SetUart0(Baudrate, PIN, OutputMode, Parity, StopBit,
                RxBufferTriggerLevel, BitMode;);
```

從[該函式說明](#)裡得知，該函式除了前兩個參數 **Baudrate** 及 **PIN** 無函式預設值之外，其餘參數皆預設值為 0，因此該函式可用下列方式簡化表示，並且捨去之參數皆會以預設值自動帶入。

```
State = SetUart0(Baudrate, PIN, OutputMode, Parity, StopBit,
                RxBufferTriggerLevel;);
State = SetUart0(Baudrate, PIN, OutputMode, Parity, StopBit;);
State = SetUart0(Baudrate, PIN, OutputMode, Parity;);
State = SetUart0(Baudrate, PIN, OutputMode;);
State = SetUart0(Baudrate, PIN;);
```

請注意，捨去之參數順序為由後往前依序簡化，若是想捨去順序在中間之參數則必須將其後方之參數也一併簡化才行。

I/O 功能腳位對應表

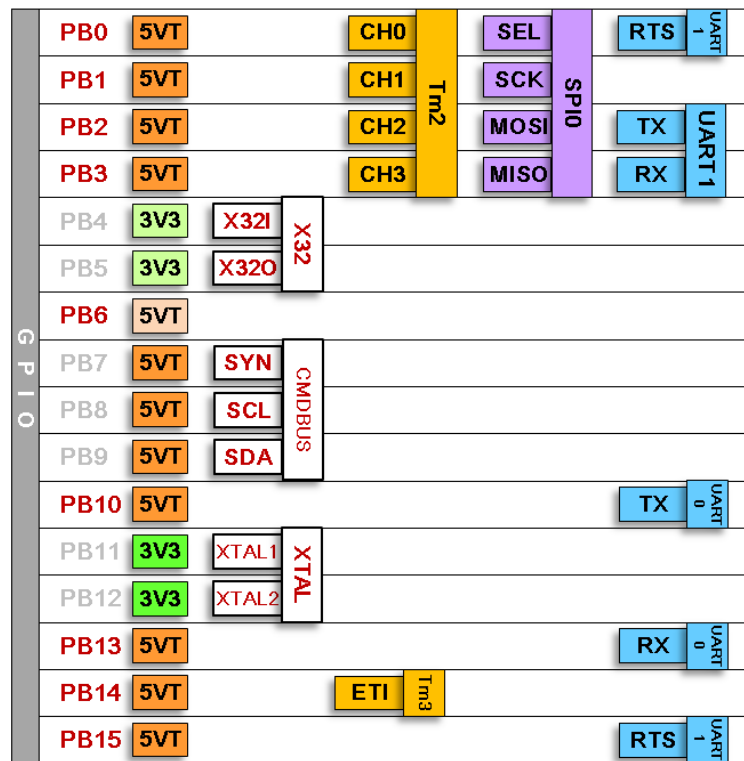
下圖為所有函式庫功能之 I/O 對應表，有些函式庫功能可不只有一對應 I/O。

Name	初始預設環境設定
3V3	1mA Drive/Sink, 3.3V電壓位準輸出/輸入
3V3	8mA Drive/Sink, 3.3V電壓位準輸出/輸入
5VT	1mA Drive/Sink, 3.3V電壓位準輸出/5V電壓位準輸入
5VT	8mA Drive/Sink, 3.3V電壓位準輸出/5V電壓位準輸入
5VT	12mA Drive/Sink, 3.3V電壓位準輸出/5V電壓位準輸入
Name	可切換功能設定
Name	
Name	
Name	
Name	
Name	

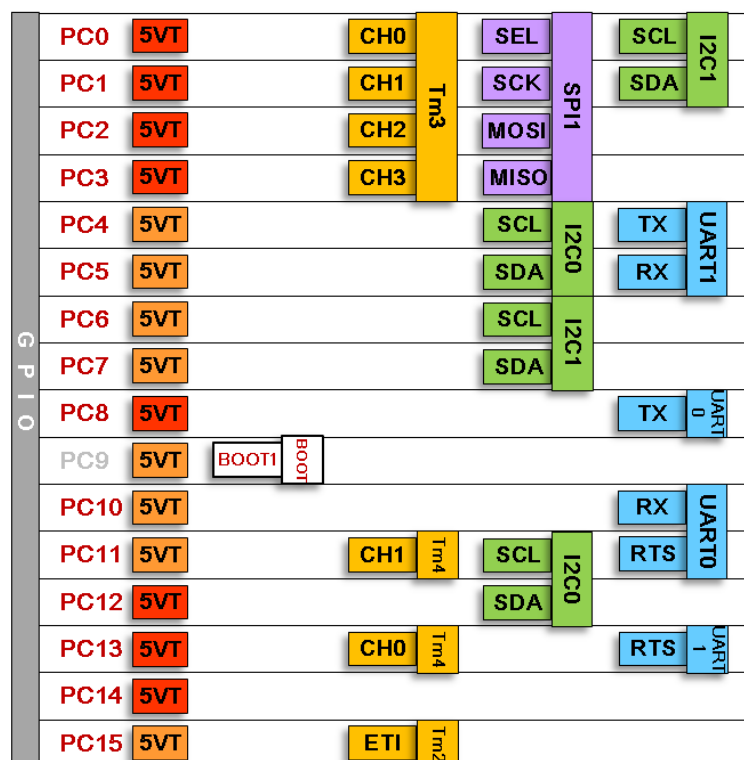
各種功能相關顏色標示說明

G P I O	PA0	3V3	AD0	ADC	CH0	Tm3					
	PA1	3V3	AD1		CH1						
	PA2	3V3	AD2		CH2				TX	UART0	
	PA3	3V3	AD3		CH3				RX	UART0	
	PA4	3V3	AD4				SEL	SPI 1	RTS	UART1	
	PA5	3V3	AD5				SCK				
	PA6	3V3	AD6				MOSI		TX	UART1	
	PA7	3V3	AD7				MISO		RX	UART1	
	PA8	5VT					SEL	SPI 1	RTS	UART0	
	PA9	5VT					SCK				
	PA10	5VT					MOSI		TX	UART0	
	PA11	5VT					MISO		RX	UART0	
	PA12	5VT			CH0	Tm3					
	PA13	5VT	LED1	LED	CH1						
	PA14	5VT	LED2		CH2						
	PA15	5VT	DETECT	USB	CH3						

Port A 對應功能圖



Port B 對應功能圖



Port C 對應功能圖

G P I O	PD0	5VT			CH0	Tm2	SEL	SPI0		SCL	I2C1
	PD1	5VT			CH1		SCK		SDA		
	PD2	5VT			CH2		MOSI				
	PD3	5VT			CH3		MISO				
	PD4	5VT					SEL	SPI1		TX	UART1
	PD5	5VT				SCK			RX		
	PD6	5VT				MOSI					
	PD7	5VT				MISO					
	PD8	5VT	CH2	Tm4	CH0	Tm3	SEL	SPI0		TX	UART0
	PD9	5VT			CH1		SCK				
	PD10	5VT	CH3	Tm4	CH2		MOSI			RX	UART0
	PD11	5VT			CH3		MISO			RTS	UART0
	PD12	5VT	ETI	Tm3	CH0	Tm4	SCL	I2C0			
	PD13	5VT	ETI	Tm2			SDA			RTS	UART1
	PD14	5VT			CH1	Tm4					
	PD15	5VT									

Port D 對應功能圖

G P I O	PE0	5VT		CH2	Tm4				
	PE1	5VT							
	PE2	5VT		CH3	Tm4				
	PE3	5VT							
	PE4	5VT							
	PE5	3V3	CN0	OPA0/CMP0	CH0	SEL	SPI1		
	PE6	3V3	CP0		CH1	SCK			
	PE7	3V3	Aout0		CH2	MOSI			
	PE8	3V3	CN1		CH3	MISO			
	PE9	3V3	CP1	OPA1/CMP1	ETI			SCL	I2C1
	PE10	3V3	Aout1		ETI	Tm3		SDA	
	PE11	5VT	SWO	IDE					
	PE12	5VT	SWCLK						
	PE13	5VT	SWDIO						
	PE14	5VT			CH0	Tm4		TX	UART1
	PE15	5VT						RX	

Port E 對應功能圖

GPIO

Arminno™提供 32GPIOs(General Purpose I/O pins)可供使用，使用特性如下：

- GPIO 分布為 PA0~15、PB0~15、PC0~15、PD0~15、PE0~15
- 除了 PB4~PB5(XTAL32 功能)、PB11~12(XTAL 功能)及 PE11~13(IDE 功能)，其他所有 I/O 皆預設為 GPIO（請參考：[I/O 功能腳位對應表](#)）
- 每一 GPIO 皆可設定為輸出/入模式(預設為輸入模式)
- 輸入有浮動(floating)/上拉(pull up)/下拉(pull down)三種組態，預設為浮動組態
- 輸出有 push pull/open drain 二種組態，預設為 push pull 組態
- 不同 GPIO 有不同 drive/sink 電流驅動能力（最小 1mA，最大 12mA，請參考所有 I/O 對應功能圖）
- 不同 GPIO 有不同容忍電壓輸入值（3.3V 或 5VT），請參考：所有 I/O 對應功能圖
- 共有 16 組可程式化外部訊號觸發事件(Event)可供設定
- PC9 務必於復位(Reset)時保持在高位準(設定啟動模式所需)

GPIO 函式庫各函式功能介紹如下：

名稱	說明
InitialGpioState()	初始設定
Low()	切換輸出模式並低位準輸出(pin:1bit)
High()	切換輸出模式並高位準輸出(pin:1bit)
Toggle()	切換輸出模式並 Toggle 輸出(pin:1bit)
In()	切換輸入模並及輸入訊號(pin:1bit)
Input()	切換輸入模式(port:16bit)
Output()	切換輸出模式(port:16bit)
ReadPort()	輸入訊號(port:16bit)
WritePort()	輸出訊號(port:16bit)
SetDirPort()	切換輸出入模式(port:16bit)
Event 相關設定	
SetGpioEvent()	啟動 GPIO 觸發 Event 0~15
SetGpioEventOff()	取消 GPIO 觸發 Event 0~15
GpioEvent0() : GpioEvent15()	GPIO 觸發 Event0 : GPIO 觸發 Event15

指令詳細說明：

unsigned char **State** = InitialGpioState(unsigned char *GPIO*,
unsigned char *InitialState*,
unsigned char *InputMode*,
unsigned char *OutputMode*);

設定 GPIO 初始化，以符合不同用途。

- **GPIO** 常數或變數值(0~79)·指定 GPIO·0~15:PA0~PA15·16~31:PB0~PB15·32~47:PC0~PC15·48~63:PD0~PD15·64~79:PE0~PE15。
- **InitialState** 常數或變數值(0~2)·設定 GPIO 初始輸出入狀態·0:input·1:output low·2:output high。
- **InputMode** 常數或變數值(0~2)·設定 GPIO 於輸入模式之上拉及下拉電阻設定·0:floating·1:pull down·2:pull up。此部份設定也用於輸出 open drain 模式之高位準上拉電阻設定(open drain 模式下 pull down 無效)·預設值為 0
- **OutputMode** 常數或變數值(0~1)·設定 GPIO 輸出驅動模式·0: push pull·1: open drain。預設值為 0
- **State** 回傳變數值(0~255)·0:設定成功·255:設定失敗。

```
#include "arminno.h"
int main(void)
{
    InitialGpioState(PA0, 1);    // PA0 設為輸出 low, push pull
    InitialGpioState(PA1, 0, 1); // PA1 設為輸入 , pull down
    InitialGpioState(PC15, 0, 2); // PC15 設為輸入 , pull up
    InitialGpioState(PD10, 1, 0, 1); // PD10 設為輸出 , open drain
    while(1);
}
```

void **Low**(unsigned char *GPIO*);

設定指定 GPIO 為輸出模式，並輸出 Low 位準。

- **GPIO** 常數或變數值(0~79)·指定 GPIO·0~15:PA0~PA15·16~31:PB0~PB15·32~47:PC0~PC15·48~63:PD0~PD15·64~79:PE0~PE15。

void **High**(unsigned char *GPIO*);

設定指定 GPIO 為輸出模式，並輸出 High 位準。

- **GPIO** 常數或變數值(0~79)·指定 GPIO·0~15:PA0~PA15·16~31:PB0~PB15·32~47:PC0~PC15·48~63:PD0~PD15·64~79:PE0~PE15。

void **Toggle**(unsigned char *GPIO*);

設定指定 GPIO 為輸出模式，並根據之前一次指定 GPIO 訊號狀態的反相輸出訊號。

- **GPIO** 常數或變數值(0~79)·指定 GPIO·0~15:PA0~PA15·16~31:PB0~PB15·32~47:PC0~PC15·48~63:PD0~PD15·64~79:PE0~PE15。

```
#include "arminno.h"
#define Led1 PA14
#define Led2 PA15
int main(void)
{
    InitialGpioState(Led1, 1);    // PA14 設為輸出 LOW
    InitialGpioState(Led2, 2);    // PA15 設為輸出 HIGH
    while(1) {
        High(Led1);               // PA14 輸出 HIGH
        Toggle(Led2);             // 切換 PA15 狀態
        Pause(5000);              // 等待 0.5 秒
        Low(Led1);                // PA14 輸出 LOW
        Toggle(Led2);             // 切換 PA15 狀態
        Pause(5000);              // 等待 0.5 秒
    }
}
```

unsigned char **State = In**(unsigned char *GPIO*);

設定指定 GPIO 為輸入模式，並回傳 GPIO 輸入訊號。

- **GPIO** 常數或變數值(0~79)·指定 GPIO·0~15:PA0~PA15·16~31:PB0~PB15·32~47:PC0~PC15·48~63:PD0~PD15·64~79:PE0~PE15。
- **State** 回傳變數值(0~1)·0:低位準信號輸入，1:高位準信號輸入。

void **Input**(unsigned char *GPIO*);

設定指定 GPIO 為輸入模式。

- **GPIO** 常數或變數值(0~79)·指定 GPIO·0~15:PA0~PA15·16~31:PB0~PB15·32~47:PC0~PC15·48~63:PD0~PD15·64~79:PE0~PE15。

void **Output**(unsigned char *GPIO*);

設定指定 GPIO 為輸出模式。

- **GPIO** 常數或變數值(0~79)·指定 GPIO·0~15:PA0~PA15·16~31:PB0~PB15·32~47:PC0~PC15·48~63:PD0~PD15·64~79:PE0~PE15。

```
#include "arminno.h"
unsigned char state;
int main(void)
{
    InitialGpioState(PA0, 0);    // PA0 設為輸入
    InitialGpioState(PA14, 1);   // PA14 設為輸出
    while(1) {
        state = In(PA0);         // 讀取 PA0 的值
        if(state == 1) Low(PA14); // 反向輸出至 PA14
        else High(PA14);
    }
}
```

unsigned short **State** = ReadPort(unsigned char **Port**);

回傳 GPIO port 輸入訊號。

- **Port** 常數或變數值(0~4) · 指定 GPIO 所屬 port 值 · 0:port A · 1:port B · 2:port C · 3:port D · 4:port E · 預設值為 0
- **State** 回傳變數值(0~65535) · bit0~bit15 資料依序為 P0~P15 之輸入資料。

void WritePort(unsigned char **Port**, unsigned short **Value**);

輸出 GPIO port 訊號。

- **Port** 常數或變數值(0~4) · 指定 GPIO 所屬 port 值 · 0:port A · 1:port B · 2:port C · 3:port D · 4:port E。
- **Value** 常數或變數值(0~65535) · bit0~bit15 資料依序為 P0~P15 之寫入資料。

void SetDirPort(unsigned char **Port**, unsigned short **Value**);

設定 GPIO port 輸出入模式。

- **Port** 常數或變數值(0~4) · 指定 GPIO 所屬 port 值 · 0:port A · 1:port B · 2:port C · 3:port D · 4:port E。
- **Value** 常數或變數值(0~65535) · bit0~bit15 資料依序為 P0~P15 之輸出入模式設定資料，該 bit 值為 0 表示為輸出模式，1 表示為輸入模式。

```
#include "arminno.h"
unsigned short state;
int main(void)
```

```

{
    SetDirPort (0, 0);           // Port A 設為輸出
    SetDirPort (1, 0xFFFF);     // Port B 設為輸入
    while(1) {
        state = ReadPort (1);    // 讀取 Port B 的值
        WritePort(0, state );    // 寫到 Port A
    }
}

```

Event 相關設定：

unsigned char ***State*** = SetGpioEvent(unsigned char ***EventNo***,
 unsigned char ***Port***, unsigned char ***Mode***,
 unsigned long ***Debounce***);

設定 GPIO Event 觸發條件並啟動，共有 16 個獨立觸發 GPIO Event 設定 (0~15)。

GPIO Event	對應腳位
GPIO Event0	PA0 或 PB0 或 PC0 或 PD0 或 PE0
GPIO Event1	PA1 或 PB1 或 PC1 或 PD1 或 PE1
GPIO Event2	PA2 或 PB2 或 PC2 或 PD2 或 PE2
GPIO Event3	PA3 或 PB3 或 PC3 或 PD3 或 PE3
GPIO Event4	PA4 或 PB4 或 PC4 或 PD4 或 PE4
GPIO Event5	PA5 或 PB5 或 PC5 或 PD5 或 PE5
GPIO Event6	PA6 或 PB6 或 PC6 或 PD6 或 PE6
GPIO Event7	PA7 或 PB7 或 PC7 或 PD7 或 PE7
GPIO Event8	PA8 或 PB8 或 PC8 或 PD8 或 PE8
GPIO Event9	PA9 或 PB9 或 PC9 或 PD9 或 PE9
GPIO Event10	PA10 或 PB10 或 PC10 或 PD10 或 PE10
GPIO Event11	PA11 或 PB11 或 PC11 或 PD11 或 PE11
GPIO Event12	PA12 或 PB12 或 PC12 或 PD12 或 PE12
GPIO Event13	PA13 或 PB13 或 PC13 或 PD13 或 PE13
GPIO Event14	PA14 或 PB14 或 PC14 或 PD14 或 PE14
GPIO Event15	PA15 或 PB15 或 PC15 或 PD15 或 PE15

- ***EventNo*** 常數或變數值(0~15)，指定所需啟動 GPIO Event 編號(0~15)。
- ***Port*** 常數或變數值(0~1)，指定 GPIO Event 所屬腳位，0:PA0~15，1:PB0~15，2:PC0~15，3:PD0~15，4:PE0~15。

- **Mode** 常數或變數值(0~4) · 設定 GPIO Event 觸發模式 · 0: 低準位 · 1: 高準位 · 2: 下降緣 · 3: 上升緣 · 4: 上升或下降緣 · 預設值為 2
- **Debounce** 常數或變數值(0~3,000,000) · 設定 GPIO Event 於低準位或高準位觸發之 Debounce 時間條件 · 0: disable · 1~3,000,000: 單位 1us
- **State** 回傳變數值(0~255) · 0:設定成功 · 255:設定失敗 · 每個 GPIO Event 皆有其對應之觸發腳位可供選擇 · 對應表如下：

unsigned char **State** = SetGpioEventOff(unsigned char **EventNo**);

取消 GPIO Event 0~15 觸發。

- **EventNo** 常數或變數值(0~15) · 指定所需取消 GPIO Event 編號(0~15)。
- **State** 回傳變數值(0~255) · 0:設定成功 · 255:設定失敗。

void GpioEvent0(void);

:

void GpioEvent15(void);

總共 16 個函式分別對應 GPIO Event0 到 Event15 · 當 GPIO Event 發生時跳至對應函式區塊執行。

```
#include "arminno.h"
unsigned int count = 0;
void GpioEvent10(void)
{
    printf("%d\r\n", count++);    // 列印偵測次數
}
int main(void)
{
    InitialGpioState(PE10, 0);    // 設為 PE10 為輸入
    SetGpioEvent(10, 4, 2, 10);   // 開啟 Event, 偵測上下緣
    while(1);
}
```

ADC

Arminno™提供 8 組高速 12bit 解析度之類比數位轉換器功能(ADC:Analog to Digital Converter)·特性如下：

- 12bit 解析度(resolution)
- 超高速 1MSPS(1,000,000 samples per second)轉換速度
- 共有 8 組 ADC 可同時使用(PA0~PA7·請參考：I/O 功能腳位對應表)
- 可接受輸入電壓為 0~3.3V

ADC 函式庫各函式使用功能介紹如下：

名稱	說明
SetAdc()	啟動 ADC 類比轉數位功能
GetAdc()	讀取 ADC 數位資料
SetAdcOff()	關閉 ADC 類比轉數位功能

指令詳細說明：

unsigned char **State** = **SetAdc**(unsigned char **Channel**);

設定 ADC Channel 並啟動。

- **Channel** 常數或變數值(0~7)·指定 ADC 啟動 Channel·0~7：AD0~7(PA0~7)。
- **State** 回傳變數值(0~255)·0: 設定成功·255: 設定失敗。

unsigned short **State** = **GetAdc**(unsigned char **Channel**);

讀取 ADC 感測值。

- **Channel** 常數或變數值(0~7)·指定 ADC 啟動 Channel·0~7：AD0~7(PA0~7)。
- **State** 回傳變數值(0~65535)·0~4095：類比轉數位回傳值·65535：資料未讀取成功。

```
#include "arminno.h"
int i;
unsigned short Value[8];
int main(void)
{
    for(i = 0 ; i < 8 ; i++) SetAdc(i);
    while(1) {
```

```
for(i = 0 ; i < 8 ; i++) Value[i] = GetAdc(i);  
printf("%4d %4d %4d %4d %4d %d %4d %4d\r\n",  
Value[0], Value[1], Value[2], Value[3],  
Value[4], Value[5], Value[6], Value[7]);  
}  
}
```

unsigned char ***State*** = SetAdcOff(unsigned char ***Channel***);

取消 ADC 功能。

- ***Channel*** 常數或變數值(0~7) · 指定 ADC 啟動 Channel · 0~7 : AD0~7(PA0~7)。
- ***State*** 回傳變數值(0~255) · 0 : 取消成功 · 255 : 取消失敗。

Timer 0

Arminno™內建基本功能硬體 Timer 0，特性如下：

- 32bit up-counting auto reload counter register
- 內部時間輸入計數功能，可產生固定時間事件(Event)

Timer 0 各函式使用功能介紹如下：

名稱	說明
產生固定時間 Event 和計數功能	
SetTm0()	設定相關參數並啟動
SetTm0Value()	設定計數初始值
GetTm0Value()	讀取計數狀態及數值
SetTm0Off()	取消設定
事件 (Event) 相關設定	
SetTm0Event()	設定 Timer overflow 觸發事件
Tm0UpdateEvent()	Timer overflow 觸發事件

Timer 相關功能設定，詳細說明如下：

產生固定時間 Event 和計數功能

使用 Timer 0 功能可利用內部時鐘 72MHz 來啟動自動計數功能，並於固定時間間隔(overflow)產生狀態通知(state)或事件(Event)，來提醒使用者於固定時間處理特定事情。Counter 計數單位為 1/72,000,000 秒

unsigned char **State** = SetTm0(unsigned long **MaxCount**);

設定 Timer 0 功能並啟動。

- **MaxCount** 常數或變數值(0~4294967295)，設定此數值可限制 Counter 最大值。預設值為 72,000,000，每 1 秒產生一次狀態通知(state)或事件(Event)
- **State** 回傳變數值(0~255)，0: 設定成功，255: 設定失敗。

unsigned char **State** = SetTm0Value(unsigned long **TmValue**);

設定 Timer 0 之數值，通常於設定初始值時使用。

- **TmValue** 常數或變數值(0~4294967295)，為直接設定 counter 數值。
- **State** 回傳變數值(0~255)，0: 設定成功，255: 設定失敗。

unsigned char **State** = GetTm0Value(unsigned long *TmValue*);

讀取 Timer 0 之數值。

- **TmValue** 回傳變數值(0~4294967295) · 讀取 counter 數值。
- **State** 回傳變數值(0~255) · 0: 無狀況 · 1: 產生 overflow updated Event · 255: 讀取失敗。

```
#include "arminno.h"
unsigned long TmValue;
int main(void)
{
    SetTm0(72000);    // 週期 1ms
    while(1) {
        GetTm0Value(TmValue);
        printf("%d\r\n", TmValue);
        Pause(5000);
    }
}
```

unsigned char **State** = SetTm0Off(void);

取消 Timer 0 設定。

- **State** 回傳變數值(0~255) · 0: 取消成功 · 255: 取消失敗。

Event 相關設定

開啟 Timer 0 Update Event 功能可於 Timer 0 計數數值產生溢位時(overflow)產生觸發事件(Event) · 使用者於該觸發事件內撰寫相關程式。

unsigned char **State** = SetTm0Event(unsigned char *UpdateEvent*);

設定 Timer 0 UpdateEvent 功能。

- **UpdateEvent** 常數或變數值(0~1) · 啟動或關閉 Update Event 功能 · 0: Disable · 1: Enable。
- **State** 回傳變數值(0~255) · 0:設定成功 · 255:設定失敗。

void Tm0UpdateEvent(void);

當發生 Tm2UpdateEvent 時(發生 overflow) · 系統所設定之程式執行區塊。

```
#include "arminno.h"
#define Led1      PA14
void Tm0UpdateEvent(void)
{
    Toggle(Led1);
}
int main(void)
{
    InitialGpioState(Led1, 1);
    SetTm0();
    SetTm0Event(1);
    while(1);
}
```

Timer 1

Arminno™內建基本功能硬體 Timer 1，特性如下：

- 32bit up-counting auto reload counter register
- 內部時間輸入計數功能，可產生固定時間事件(Event)

Timer 1 各函式使用功能介紹如下：

名稱	說明
產生固定時間 Event 和計數功能	
SetTm1()	設定相關參數並啟動
SetTm1Value()	設定計數初始值
GetTm1Value()	讀取計數狀態及數值
SetTm1Off()	取消設定
事件 (Event) 相關設定	
SetTm1Event()	設定 Timer overflow 觸發事件
Tm1UpdateEvent()	Timer overflow 觸發事件

Timer 相關功能設定，詳細說明如下：

產生固定時間 Event 和計數功能

使用 Timer 1 功能可利用內部時鐘 72MHz 來啟度自動計數功能，並於固定時間間隔(overflow)產生狀態通知(state)或事件(Event)，來提醒使用者於固定時間處理特定事情。Counter 計數單位為 1/72,000,000 秒

unsigned char **State** = SetTm1(unsigned long *MaxCount*);

設定 Timer 1 功能並啟動。

- **MaxCount** 常數或變數值(0~4294967295)，設定此數值可限制 Counter 最大值。預設值為 72,000,000，每 1 秒產生一次狀態通知(state)或事件(Event)
- **State** 回傳變數值(0~255)，0: 設定成功，255: 設定失敗。

unsigned char **State** = SetTm1Value(unsigned long *TmValue*);

設定 Timer 1 之數值，通常於設定初始值時使用。

- **TmValue** 常數或變數值(0~4294967295)，為直接設定 Counter 數值。
- **State** 回傳變數值(0~255)，0: 設定成功，255: 設定失敗。

unsigned char **State** = GetTm1Value(unsigned long *TmValue*);

讀取 Timer 1 之數值。

- **TmValue** 回傳變數值(0~4294967295) · 讀取 Counter 數值。
- **State** 回傳變數值(0~255) · 0: 無狀況 · 1: 產生 overflow updated Event · 255: 讀取失敗。

```
#include "arminno.h"
#define Led2      PA15
#define k05Sec    (72*500000)
unsigned long TmValue;
int main(void)
{
    InitialGpioState(Led2, 1);
    SetTm1();
    while(1) {
        GetTm1Value(TmValue);
        if(TmValue >= k05Sec) {
            SetTm1Value(0);
            Toggle(Led2);
        }
    }
}
```

unsigned char **State** = SetTm1Off(void);

取消 Timer 1 設定。

- **State** 回傳變數值(0~255) · 0: 取消成功 · 255: 取消失敗。

Event 相關設定

開啟 Timer 1 Update Event 功能可於 Timer 1 計數數值產生溢位時(overflow)產生觸發事件(Event) · 使用者於該觸發事件內撰寫相關程式。

unsigned char **State** = SetTm1Event(unsigned char *UpdateEvent*);

設定 Timer 1 UpdateEvent 功能。

- **UpdateEvent** 常數或變數值(0~1) · 啟動或關閉 Update Event 功能 · 0: Disable · 1: Enable。
- **State** 回傳變數值(0~255) · 0:設定成功 · 255:設定失敗。

`void Tm1UpdateEvent(void);`

當發生 Tm1UpdateEvent 時(發生 overflow)・系統所設定之程式執行區塊。

Timer 2

Arminno™內建多功能硬體 Timer 2，特性如下：

- 16bit up/down auto reload counter register
- 內部/外部時間輸入計數功能，可產生固定時間事件(Event)
- 擁有 4 Channel 同時輸出之可程式化 PWM(Pulse Width Modulation)功能
- 可切換三組不同輸出入 I/O (請參考：[I/O 功能腳位對應表](#))
- 單一脈波輸入量測功能
- 單一脈波可調寬度輸出功能
- 二線式編碼器解碼功能
- Counter 數值溢位(underflow/overflow)事件觸發(Event trigger)功能

Timer 2 各函式使用功能介紹如下：

名稱	說明
產生固定時間 Event 和計數功能	
SetTm2Counter()	設定 Counter 相關參數並啟動
SetTm2CounterValue()	設定 Counter 計數初始值
GetTm2CounterValue()	讀取 Counter 計數狀態及數值
SetTm2CounterOff()	取消 Counter 設定
輸出 PWM 訊號功能	
SetTm2Pwm()	設定 PWM 相關參數並啟動
SetTm2PwmCh()	啟動及關閉 PWM channel 輸出
SetTm2PwmOff()	取消 PWM 設定
單一輸入脈波寬度量測功能	
SetTm2PulseIn()	設定 PulseIn 相關參數並啟動
GetTm2PulseInValue()	讀取 PulseIn 量測狀態及數值
SetTm2PulseInOff()	取消 PulseIn 設定
單一脈波輸出功能	
SetTm2PulseOut()	設定 PulseOut 相關參數並啟動
GetTm2PulseOutState()	讀取 PulseOut 輸出狀態
SetTm2PulseOutOff()	取消 PulseOut 設定
頻率輸出功能	
SetTm2FreqOut()	設定 FreqOut 相關參數並啟動
SetTm2FreqOutOff()	取消 FreqOut 設定
編碼器解碼功能	
SetTm2Decoder()	設定 Decoder 相關參數並啟動

SetTm2DecoderValue()	設定 Decoder 計數初始值
GetTm2DecoderValue()	讀取 Decoder 計數值
SetTm2DecoderOff()	取消 Decoder 設定
事件 (Event) 相關設定	
SetTm2Event()	設定 Timer overflow/underflow 觸發事件
Tm2UpdateEvent()	Timer overflow/underflow 觸發事件

Timer 相關功能設定，使用時在同一時間裡只能啟動其中一項功能，若要啟動不同的功能，則須先將之前功能設定取消後再啟動另一功能，詳細說明如下：

產生固定時間 Event 和計數功能

使用 Timer 2 counter 功能除了可利用內外部時鐘輸入來設定自動計數功能之外，更可讓 Arminno™於程式執行期間，於固定時間間隔(overflow or underflow)產生狀態通知(state)或事件(Event)，來提醒使用者於固定時間處理特定事情。

```
unsigned char State = SetTm2Counter( unsigned char InputMode,
                                     unsigned char CountMode,
                                     unsigned short Prescaler,
                                     unsigned short MaxCount);
```

設定 Timer 2 counter 功能並啟動。

- **InputMode** 常數或變數值(0~4)，選擇 Counter clock 來源。0: 內部系統時鐘 Fclk(72MHz) · 1: 外部時鐘輸入(PB7) · 2: 外部時鐘輸入(PC15) · 3: 外部時鐘輸入(PD13) · 4: 外部時鐘輸入(PE9)。預設值為 0

<i>InputMode</i>	時鐘輸入源
0	內部系統時鐘 Fclk(72MHz)
1	外部時鐘輸入(PB7)
2	外部時鐘輸入(PC15)
3	外部時鐘輸入(PD13)
4	外部時鐘輸入(PE9)

- **CountMode** 常數或變數值(0~2)，Counter 計數方式，0:遞增(up) · 1: 遞減(down) · 2: 遞增+遞減循環(center)。預設值為 0
- **Prescaler** 常數或變數值(0~65535)，可將輸入之時鐘頻率除頻，進而調整計數速度，公式為： 頻率=內部(外部)時鐘輸入/(Prescaler+1)。預設值為 719，若為 72MHz 內部時鐘輸入時，除頻之輸入時鐘單位為 1/100,000 秒
- **MaxCount** 常數或變數值(0~65535)，設定此數值可限制 Counter 最大值。預設值為 10000，配合預設 Prescaler 則為每 0.1 秒產生狀態通知(state)或事件

(Event)

- **State** 回傳變數值(0~255) · 0: 設定成功 · 255: 設定失敗。

unsigned char **State** = **SetTm2CounterValue**(unsigned short **TmValue**);

設定 Timer 2 counter 之數值，通常於設定初始值時使用。

- **TmValue** 常數或變數值(0~65535)，為直接設定 Counter 數值。
- **State** 回傳變數值(0~255) · 0: 設定成功 · 255: 設定失敗。

unsigned char **State** = **GetTm2CounterValue**(unsigned short **TmValue**);

讀取 Timer 2 counter 之數值。

- **TmValue** 回傳變數值(0~65535)，讀取 Counter 數值。
- **State** 回傳變數值(0~255) · 0: 無狀況 · 1: 產生 updated Event(overflow or underflow) · 255: 讀取失敗。

```
#include "arminno.h"
#define Led2      PA15
unsigned long TmValue;
void Tm2UpdateEvent(void)
{
    Toggle(Led2);
}
int main(void)
{
    InitialGpioState(Led2, 1);           // 設定 PA15 輸出
    SetTm2Counter (0, 0, 719, 50000);    // 設定 TM2 週期為 0.5 秒
    SetTm2Event(1);                      // 啟動 Event 功能
    while(1);
}
```

unsigned char **State** = **SetTm2CounterOff**(void);

取消 Timer 2 counter 之功能設定。

- **State** 回傳變數值(0~255) · 0: 取消成功 · 255: 取消失敗。

輸出 PWM 訊號功能

使用 Timer 2 PWM 的功能除了可以透過自訂 PWM 頻率及 PWM 解析度功能來達成所需要的 PWM 波形之外，更可藉由最多四個 Channel 輸出的功能來達到多元件同步控制的效果。

unsigned char **State** = SetTm2Pwm(unsigned short **Prescaler**,
unsigned short **Resolution**);

設定 Timer 2 PWM 功能並啟動。

- **Prescaler** 常數或變數值(0~65535)，可將輸入之時鐘頻率做除頻動作，進而調整 PWM 每單位時間速度，公式如下：速度=72MHz/(Prescaler+1)，預設值為 71，若為 72MHz 內部時鐘輸入時，除頻之輸入時鐘單位為 1/1,000,000 秒
- **Resolution** 常數或變數值(0~65535)，為直接設定 PWM 可用解析度範圍數值。預設值為 1000，配合預設 Prescaler 則為產生解析度為 1000 單位之 1KHz PWM 波型輸出
- **State** 回傳變數值(0~255)，0:設定成功，255:設定失敗。

unsigned char **State** = SetTm2PwmCh(unsigned char **Channel**,
unsigned char **Enable**, unsigned char **Mode**,
unsigned short **Duty**);

設定 Timer 2 PWM 所要輸出之 Channel 設定。

- **Channel** 常數或變數值(0~3)，選擇 PWM Channel。Channel 選擇請參考下表之對應。
- **Enable** 常數或變數值(0~3)，0: 關閉輸出，1:開啟輸出 1 (Ch0: PB0、Ch1: PB1、Ch2: PB2 和 Ch3: PB3)，2:開啟輸出 2 (Ch0: PD0、Ch1: PD1、Ch2: PD2 和 Ch3: PD3)，3:開啟輸出 3 (Ch0: PE5、Ch1: PE6、Ch2: PE7 和 Ch3: PE8)。預設值為 0

Enable	Ch0	Ch1	CH2	Ch3
0	關閉輸出	關閉輸出	關閉輸出	關閉輸出
1	PB0	PB1	PB2	PB3
2	PD0	PD1	PD2	PD3
3	PE5	PE6	PE7	PE8

- **Mode** 常數或變數值(0~1)，0:push pull，1:open drain。預設值為 0
- **Duty** 常數或變數值(0~65535)，設定 PWM Channel 之比例輸出值。預設值為 0
- **State** 回傳變數值(0~255)，0: 設定成功，255: 設定失敗。

```
#include "arminno.h"
int main(void)
{
    SetTm2Pwm ();           // 週期 1ms
    while(1) {
        SetTm2PwmCh(0, 1, 0, 100);    // PB0, duty = 100us
    }
}
```

```
SetTm2PwmCh(1, 1, 0, 200);    // PB1, duty = 200us
SetTm2PwmCh(2, 1, 0, 300);    // PB2, duty = 300us
SetTm2PwmCh(3, 1, 0, 400);    // PB3, duty = 400us
Pause(20000);                  // 延遲 2 秒
SetTm2PwmCh(0, 1, 0, 900);    // PB0, duty = 900us
SetTm2PwmCh(1, 1, 0, 800);    // PB1, duty = 800us
SetTm2PwmCh(2, 1, 0, 700);    // PB2, duty = 700us
SetTm2PwmCh(3, 1, 0, 600);    // PB3, duty = 600us
Pause(20000);                  // 延遲 2 秒
}
}
```

unsigned char **State** = SetTm2PwmOff(void)

取消 Timer 2 PWM 之功能設定。

- **State** 回傳變數值(0~255) · 0: 取消成功 · 255: 取消失敗。

單一輸入脈波寬度量測功能

Timer 2 Pulse In 功能可以完全精準的量測所要輸入的方波波形寬度，並可自訂量測刻度大小。

unsigned char **State** = SetTm2PulseIn(unsigned char **Input**,
unsigned char **Trigger**, unsigned short **Prescaler**);

設定 Timer 2 Pulse In 功能並啟動。

- **Input** 常數或變數值(0~5) · 選擇不同輸入腳位 · 0: PB0 · 1: PB1 · 2: PD0 · 3: PD1 · 4: PE5 · 5: PE6。預設值為 0

<i>Input</i>	輸入腳位
0	PB0
1	PB1
2	PD0
3	PD1
4	PE5
5	PE6

- **Trigger** 常數或變數值(0~1) · 所要量測 Pulse 類型 · 0: Low pulse 及 1: High pulse。預設值為 0
- **Prescaler** 常數或變數值(0~65535) · 可將輸入 Timer 2 之時鐘頻率除頻，進而

設定 PulseIn 寬度量測可用單位大小，公式如下：PulseIn 量測頻率
 $= 72\text{MHz} / (\text{Prescaler} + 1)$ ，預設值為 71，依照預設值設定則脈波輸入量測單位為
 1/1,000,000 秒(uS)

- **State** 回傳變數值(0~255)，0:設定成功，255:設定失敗。

unsigned char **State** = GetTm2PulseInValue(unsigned Short **Value**);

讀取 Timer 2 PulseIn 量測數值和狀態。

- **Value** 回傳變數值(0~65535)，PulseIn 量測數值回傳。
- **State** 回傳值(0~255)，回傳目前 PulseIn 量測狀況，0: 設定成功，1: 等待 PulseIn 量測完成，2: 超過所能量測最大值(overflow)，255: 讀取失敗。

```
#include "arminno.h"
int main(void)
{
    unsigned short Value;
    unsigned char status;
    SetTm2PulseIn (2);                // PD0
    do {
        status = GetTm2PulseInValue(Value);
    } while(status == 1);              // 等待測量完成
    if(status == 0) {
        printf("Pulse In = %d\r\n", Value); // 成功
    } else if(status == 2) {
        printf("Overflow\r\n");           // 溢位
    } else {
        printf("Error\r\n");              // 錯誤
    }
}
```

unsigned char **State** = SetTm2PulseInOff(void)

取消 Timer 2 PulseIn 功能設定。

- **State** 回傳變數值(0~255)，0: 取消成功，255: 取消失敗。

單一脈波輸出功能

Timer 2 Pulse Out 功能可針對不同使用設定，輸出一設定寬度之方波訊號

unsigned char **State** = SetTm2PulseOut(unsigned char **Output**,
 unsigned char **Width**, unsigned short **Level**,

unsigned short *Prescaler*);

設定 Timer 2 Pulse Out 功能並啟動。

- **Output** 常數或變數值(0~11)，選擇不同輸出腳位，0: PB0、1: PB1、2: PB2、3: PB3、4: PD0、5: PD1、6: PD2 及 7: PD3、8: PE5、9: PE6、10: PE7 及 11: PE8。
預設值為 0

<i>Input</i>	輸入腳位	<i>Input</i>	輸入腳位
0	PB0	6	PD2
1	PB1	7	PD3
2	PB2	8	PE5
3	PB3	9	PE6
4	PD0	10	PE7
5	PD1	11	PE8

- **Width** 常數或變數值(0~65535)，選擇要輸出 Pulse 寬度。預設值為 1000
- **Level** 常數或變數值(0~1)，所要輸出 Pulse 類型，0: Low pulse 及 1: High pulse。
預設值為 1
- **Prescaler** 常數或變數值(0~65535)，將輸入 Timer 2 之時鐘頻率除頻，設定 PulseOut 寬度輸出可用單位大小，公式如下：PulseOut 輸出單位頻率
=72MHz/(Prescaler+1)，預設值為 71，依照預設值設定則脈波輸出最小單位為
1/1,000,000 秒(uS)
- **State** 回傳變數值(0~255)，0:設定成功，255:設定失敗。

unsigned char *State* = GetTm2PulseOutState(void);

讀取 Timer 2 PulseOut 輸出狀態。

- **State** 回傳變數值(0~255)，回傳目前 PulseOut 輸出狀況，0: 輸出成功或無狀況，1: 等待 PulseOut 輸出完成，255: 讀取失敗。

```
#include "arminno.h"
int main(void)
{
    InitialGpioState(PB0, 1);
    SetTm2PulseOut(0); // PB0, 1000us high pulse
    while(GetTm2PulseOutState() == 1);
    while(1);
}
```

unsigned char **State** = SetTm2PulseOutOff(void)

取消 Timer 2 PulseOut 功能設定。

- **State** 回傳變數值(0~255) · 0: 取消成功 · 255: 取消失敗。

頻率輸出功能

Timer 2 Frequency Out 功能可以輸出從 1Hz~1MHZ 之間的不同頻率之方波。

unsigned char **State** = SetTm2FreqOut(unsigned char **Output**,
unsigned long **Frequency**, unsigned char **Mode**);

設定 Timer 2 Frequency Out 功能並啟動。

- **Output** 常數或變數值(0~11) · 選擇不同輸出腳位 · 0: PB0 · 1: PB1 · 2: PB2 · 3: PB3 · 4: PD0 · 5: PD1 · 6: PD2 及 7: PD3 · 8: PE5 · 9: PE6 · 10: PE7 及 11: PE8 · 預設值為 0

<i>Output</i>	輸出腳位	<i>Output</i>	輸出腳位
0	PB0	6	PD2
1	PB1	7	PD3
2	PB2	8	PE5
3	PB3	9	PE6
4	PD0	10	PE7
5	PD1	11	PE8

- **Frequency** 常數或變數值(1~1,000,000) · 可選擇 1Hz~1MHz 的頻率輸出。預設值為 1
- **Mode** 常數或變數值(0~1) · 設定輸出狀態 · 0:push pull · 1:open drain · 預設值為 0
- **State** 回傳值(0~255) · 0:設定成功 · 255:設定失敗。

```
#include "arminno.h"
int main(void)
{
    SetTm2FreqOut(0, 1000); // PB0, 輸出 1KHz
    while(1);
}
```

unsigned char **State** = SetTm2FreqOutOff(void);

取消 Timer 2 Frequency Out 輸出設定。

- **State** 回傳變數值(0~255) · 0: 取消成功 · 255: 取消失敗。

編碼器解碼功能

Timer 2 Decoder 功能可以讀取任何二線式編碼器(2-pin encoder)回授訊號。

unsigned char **State** = **SetTm2Decoder**(unsigned char **Input**,
 unsigned short **Prescaler**,
 unsigned short **MaxCount**);

設定 Timer 2 Decoder 功能並啟動。

- **Input** 常數或變數值(0~2) · 選擇不同輸入腳位 · 0:PB0 和 PB1 · 1:PD0 和 PD1 · 2:PE5 和 PE6 · 預設值為 0

Input	輸入腳位
0	PB0 和 PB1
1	PD0 和 PD1
2	PE5 和 PE6

- **Prescaler** 常數或變數值(0~65535) · 可將輸入 Timer 2 之編碼器頻率除頻 · 設定 Timer 2 Decoder 可用單位(解析度)大小 · 公式如下: Timer 2 Decoder count 數值=外部編碼器輸入 count 數值/(Prescaler+1) · 預設值為 0
- **MaxCount** 常數或變數值(0~65535) · 選擇 Decoder count 最大值 · 超過最大值即產生溢位(overflow event) · 預設值為 65535
- **State** 回傳值(0~255) · 0:設定成功 · 255:設定失敗。

unsigned char **State** = **SetTm2DecoderValue**(short **TmValue**);

設定 Timer 2 Decoder 計數值。

- **TmValue** 常數和變數值(-32768~32767) · 設定 Timer 2 Decoder count 計數值 · 預設值為 0
- **State** 回傳變數值(0~255) · 0:無狀況 · 255:讀取失敗。

unsigned char **State** = **GetTm2DecoderValue**(short **TmValue**);

讀取 Timer 2 Decoder 計數值及狀態。

- **TmValue** 回傳變數值(-32768~32767) · 回傳 Timer 2 Decoder count 計數值。
- **State** 回傳變數值(0~255) · 0:無狀況 · 1:發生溢位(overflow or underflow) · 255:讀取失敗。

```

#include "arminno.h"
short TmValue;
int main(void)
{
    SetTm2Decoder (); // PB0, PB1
    while(1) {
        GetTm2DecoderValue(TmValue); // 取得 Encoder 的值
        printf("\r%d\r\n", TmValue);
        Pause(2000);
    }
}

```

unsigned char ***State*** = SetTm2DecoderOff(void);

取消 Timer 2 Decoder 設定。

- ***State*** 回傳變數值(0~255) · 0: 取消成功 · 255: 取消失敗。

Event 相關設定

開啟 Timer 2 Update Event 功能可於 Timer 2 計數數值產生溢位時(overflow or underflow)產生觸發事件(Event)，使用者於該觸發事件內撰寫相關程式。

unsigned char ***State*** = SetTm2Event(unsigned char ***UpdateEvent***);

設定 Timer 2 UpdateEvent 功能。

- ***UpdateEvent*** 常數或變數值(0~1) · 啟動或關閉 Update Event 功能 · 0: Disable · 1: Enable。
- ***State*** 回傳變數值(0~255) · 0:設定成功 · 255:設定失敗。

void Tm2UpdateEvent(void);

當發生 Tm2UpdateEvent 時(發生 overflow or underflow)，系統所設定之程式執行區塊。

Timer 3

Arminno™內建多功能硬體 Timer 3，特性如下：

- 16bit up/down auto reload counter register
- 內部/外部時間輸入計數功能，可產生固定時間事件(Event)
- 擁有 4 Channel 同時輸出之可程式化 PWM(Pulse Width Modulation)功能
- 可切換四組不同輸出入 I/O (請參考：[I/O 功能腳位對應表](#))
- 單一脈波輸入量測功能
- 單一脈波可調寬度輸出功能
- 二線式編碼器解碼功能
- Counter 數值溢位(underflow/overflow)事件觸發(Event trigger)功能

Timer 3 各函式使用功能介紹如下：

名稱	說明
產生固定時間 Event 和計數功能	
SetTm3Counter()	設定 Counter 相關參數並啟動
SetTm3CounterValue()	設定 Counter 計數初始值
GetTm3CounterValue()	讀取 Counter 計數狀態及數值
SetTm3CounterOff()	取消 Counter 設定
輸出 PWM 訊號功能	
SetTm3Pwm()	設定 PWM 相關參數並啟動
SetTm3PwmCh()	啟動及關閉 PWM channel 輸出
SetTm3PwmOff()	取消 PWM 設定
單一輸入脈波寬度量測功能	
SetTm3PulseIn()	設定 PulseIn 相關參數並啟動
GetTm3PulseInValue()	讀取 PulseIn 量測狀態及數值
SetTm3PulseInOff()	取消 PulseIn 設定
單一脈波輸出功能	
SetTm3PulseOut()	設定 PulseOut 相關參數並啟動
GetTm3PulseOutState()	讀取 PulseOut 輸出狀態
SetTm3PulseOutOff()	取消 PulseOut 設定
頻率輸出功能	
SetTm3FreqOut()	設定 FreqOut 相關參數並啟動
SetTm3FreqOutOff()	取消 FreqOut 設定
編碼器解碼功能	
SetTm3Decoder()	設定 Decoder 相關參數並啟動

SetTm3DecoderValue()	設定 Decoder 計數初始值
GetTm3DecoderValue()	讀取 Decoder 計數值
SetTm3DecoderOff()	取消 Decoder 設定
事件 (Event) 相關設定	
SetTm3Event()	設定 Timer overflow/underflow 觸發事件
Tm3UpdateEvent()	Timer overflow/underflow 觸發事件

Timer 相關功能設定，使用時在同一時間裡只能啟動其中一項功能，若要啟動不同的功能，則須先將之前功能設定取消後再啟動另一功能，詳細說明如下：

產生固定時間 Event 和計數功能

使用 Timer 3 counter 功能除了可利用內外部時鐘輸入來設定自動計數功能之外，更可讓 Arminno™於程式執行期間，於固定時間間隔(overflow or underflow)產生狀態通知(state)或事件(Event)，來提醒使用者於固定時間處理特定事情。

```
unsigned char State = SetTm3Counter( unsigned char InputMode,
                                     unsigned char CountMode,
                                     unsigned short Prescaler,
                                     unsigned short MaxCount);
```

設定 Timer 3 counter 功能並啟動。

- **InputMode** 常數或變數值(0~3)，選擇 Counter clock 來源。0: 內部系統時鐘 Fclk(72MHz) · 1: 外部時鐘輸入(PB14) · 2: 外部時鐘輸入(PD12) · 3:外部時鐘輸入(PE10)。預設值為 0

InputMode	時鐘輸入源
0	內部系統時鐘 Fclk(72MHz)
1	外部時鐘輸入(PB14)
2	外部時鐘輸入(PD12)
3	外部時鐘輸入(PE10)

- **CountMode** 常數或變數值(0~2)，Counter 計數方式。0:遞增(up) · 1: 遞減(down) · 2: 遞增+遞減循環(center)。預設值為 0
- **Prescaler** 常數或變數值(0~65535)，可將輸入之時鐘頻率除頻，進而調整計數速度，公式為： $\text{頻率} = \text{內部(外部)時鐘輸入} / (\text{Prescaler} + 1)$ 。預設值為 719，若為 72MHz 內部時鐘輸入時，除頻之輸入時鐘單位為 1/100,000 秒
- **MaxCount** 常數或變數值(0~65535)，設定此數值可限制 Counter 最大值。預設值為 10000，配合預設 Prescaler 則為每 0.1 秒產生狀態通知(state)或事件

(Event)

- **State** 回傳變數值(0~255) · 0: 設定成功 · 255: 設定失敗。

unsigned char **State** = **SetTm3CounterValue**(unsigned short **TmValue**);

設定 Timer 3 counter 之數值，通常於設定初始值時使用。

- **TmValue** 常數或變數值(0~65535)，為直接設定 Counter 數值。
- **State** 回傳變數值(0~255) · 0: 設定成功 · 255: 設定失敗。

unsigned char **State** = **GetTm3CounterValue**(unsigned short **TmValue**);

讀取 Timer 3 counter 之數值。

- **TmValue** 回傳變數值(0~65535)，讀取 Counter 數值。
- **State** 回傳變數值(0~255) · 0: 無狀況 · 1: 產生 updated Event(overflow or underflow) · 255: 讀取失敗。

```
#include "arminno.h"
#define Led2      PA15
unsigned long TmValue;
void Tm3UpdateEvent(void)
{
    Toggle(Led2);
}
int main(void)
{
    InitialGpioState(Led2, 1);
    SetTm3Counter (0, 0, 719, 25000); // 設定 TM3 週期 0.25 秒
    SetTm3Event(1);                  // 開啟 Event
    while(1);
}
```

unsigned char **State** = **SetTm3CounterOff**(void);

取消 Timer 3 counter 之功能設定。

- **State** 回傳變數值(0~255) · 0: 取消成功 · 255: 取消失敗。

輸出 PWM 訊號功能

使用 Timer 1 PWM 的功能除了可以透過自訂 PWM 頻率及 PWM 解析度功能來達成所需要的 PWM 波形之外，更可藉由最多四個 Channel 輸出的功能來達到多元件同步

控制的效果。

```
unsigned char State = SetTm3Pwm(unsigned short Prescaler,  
                                unsigned short Resolution);
```

設定 Timer 3 PWM 功能並啟動。

- **Prescaler** 常數或變數值(0~65535)，可將輸入之時鐘頻率做除頻動作，進而調整 PWM 每單位時間速度，公式如下：速度=72MHz/(Prescaler+1)，預設值為 71，若為 72MHz 內部時鐘輸入時，除頻之輸入時鐘單位為 1/1,000,000 秒
- **Resolution** 常數或變數值(0~65535)，為直接設定 PWM 可用解析度範圍數值。預設值為 1000，配合預設 Prescaler 則為產生解析度為 1000 單位之 1KHz PWM 波型輸出
- **State** 回傳變數值(0~255)，0:設定成功，255:設定失敗。

```
unsigned char State = SetTm3PwmCh( unsigned char Channel,  
                                unsigned char Enable, unsigned char Mode,  
                                unsigned short Duty);
```

設定 Timer 3 PWM 所要輸出之 Channel 設定。

- **Channel** 常數或變數值(0~3)，選擇 PWM Channel。Channel 選擇請參考下表之對應。
- **Enable** 常數或變數值(0~4)，0: 關閉輸出，1:開啟輸出 1(Ch0:PA0、Ch1:PA1、Ch2:PA2 或 Ch3:PA3)，2:開啟輸出 2(Ch0:PA12、Ch1:PA13、Ch2:PA14 或 Ch3:PA15)，3:開啟輸出 3(Ch0:PC0、Ch1:PC1、Ch2:PC2 或 Ch3:PC3)，4:開啟輸出 4(Ch0:PD8、Ch1:PD9、Ch2:PD10 或 Ch3:PD11)。預設值為 0

Enable	Ch0	Ch1	CH2	Ch3
0	關閉輸出	關閉輸出	關閉輸出	關閉輸出
1	PA0	PA1	PA2	PA3
2	PA12	PA13	PA14	PA15
3	PC0	PC1	PC2	PC3
4	PD8	PD9	PD10	PD11

- **Mode** 常數或變數值(0~1)，0:push pull，1:open drain。預設值為 0
- **Duty** 常數或變數值(0~65535)，設定 PWM Channel 之比例輸出值。預設值為 0
- **State** 回傳變數值(0~255)，0: 設定成功，255: 設定失敗。
-

```
#include "arminno.h"  
int main(void)
```

```

{
    SetTm3Pwm(71, 20000);          // period 20 ms
    while(1) {
        SetTm3PwmCh(0, 1, 0, 700); // PA0, duty=700us
        SetTm3PwmCh(1, 1, 0, 800); // PA1, duty=800us
        SetTm3PwmCh(2, 1, 0, 900); // PA2, duty=900us
        SetTm3PwmCh(3, 1, 0, 1000); // PA3, duty=1ms
        Pause(20000);              // 延遲 2 秒
        SetTm3PwmCh(0, 1, 0, 1800); // PA0, duty=1.8ms
        SetTm3PwmCh(1, 1, 0, 1900); // PA1, duty=1.9ms
        SetTm3PwmCh(2, 1, 0, 2000); // PA2, duty=2.0ms
        SetTm3PwmCh(3, 1, 0, 2100); // PA3, duty=2.1ms
        Pause(20000);              // 延遲 2 秒
    }
}

```

unsigned char **State** = SetTm3PwmOff(void)

取消 Timer 3 PWM 之功能設定。

- **State** 回傳變數值(0~255) · 0:設定成功 · 255:設定失敗。

單一輸入脈波寬度量測功能

Timer 3 Pulse In 功能可以完全精準的量測所要輸入的方波波形寬度，並可自訂量測刻度大小。

unsigned char **State** = SetTm3PulseIn(unsigned char **Input**,
unsigned char **Trigger**, unsigned short **Prescaler**);

設定 Timer 3 Pulse In 功能並啟動。

- **Input** 常數或變數值(0~7) · 選擇不同輸入腳位 · 0:PA0 · 1:PA1 · 4:PA12 · 5:PA13 · 8:PC0 · 9:PC1 · 12:PD8 · 13:PD9 · 預設值為 0

Input	輸入腳位	Input	輸入腳位
0	PA0	4	PC0
1	PA1	5	PC1
2	PA12	6	PD8
3	PA13	7	PD9

- **Trigger** 常數或變數值(0~1) · 所要量測 Pulse 類型 · 0:Low pulse 及 1:High pulse · 預設值為 0

- **Prescaler** 常數或變數值(0~65535) · 可將輸入 Timer1 之時鐘頻率除頻 · 進而設定 PulseIn 寬度量測可用單位大小 · 公式如下：PulseIn 量測頻率 = $72\text{MHz}/(\text{Prescaler}+1)$ · 預設值為 71 · 依照預設值設定則脈波輸出最小單位為 1/1,000,000 秒(uS)
- **State** 回傳變數值(0~255) · 0:設定成功 · 255:設定失敗。

unsigned char **State** = GetTm3PulseInValue(unsigned Short **Value**);

讀取 Timer 3 PulseIn 量測數值和狀態。

- **Value** 回傳變數值(0~65535) · PulseIn 量測數值回傳。
- **State** 回傳值(0~255) · 回傳目前 PulseIn 量測狀況 · 0: 設定成功 · 1: 等待 PulseIn 量測完成 · 2: 超過所能量測最大值(overflow) · 255: 讀取失敗。

```
#include "arminno.h"
int main(void)
{
    unsigned short Value;
    unsigned char status;
    SetTm3PulseIn (4);    // PC0
    do {
        status = GetTm3PulseInValue(Value);
    } while(status == 1);    // 取樣是否完成
    if(status == 0) {
        printf("Pulse In = %d\r\n", Value); // 成功
    } else if(status == 2) {
        printf("Overflow\r\n");           // 溢位
    } else {
        printf("Error\r\n");              // 失敗
    }
}
```

unsigned char **State** = SetTm3PulseInOff(void)

取消 Timer 3 PulseIn 功能設定。

- **State** 回傳變數值(0~255) · 0:設定成功 · 255:設定失敗。

單一脈波輸出功能

Timer 3 Pulse Out 功能可針對不同使用設定 · 輸出一設定寬度之方波訊號

unsigned char **State** = SetTm3PulseOut(unsigned char **Output**,
 unsigned char **Width**, unsigned short **Level**,
 unsigned short **Prescaler**);

設定 Timer 3 Pulse Out 功能並啟動。

- **Output** 常數或變數值(0~15)，選擇不同輸出腳位，0:PA0，1:PA1，2:PA2，3:PA3，4:PA12，5:PA13，6:PA14，7:PA15，8:PC0，9:PC1，10:PC2，11:PC3，12:PD8，13:PD9，14:PD10，15:PD11。預設值為 0

Output	輸出腳位	Output	輸出腳位
0	PA0	8	PC0
1	PA1	9	PC1
2	PA2	10	PC2
3	PA3	11	PC3
4	PA12	12	PD8
5	PA13	13	PD9
6	PA14	14	PD10
7	PA15	15	PD11

- **Width** 常數或變數值(0~65535)，選擇要輸出 Pulse 寬度。預設值為 1000
- **Level** 常數或變數值(0~1)，所要輸出 Pulse 類型，0: Low pulse 及 1: High pulse。預設值為 1
- **Prescaler** 常數或變數值(0~65535)，將輸入 Timer 3 之時鐘頻率除頻，設定 PulseOut 寬度輸出可用單位大小，公式如下：PulseOut 輸出單位頻率 = 72MHz/(Prescaler+1)，預設值為 71，依照預設值設定則脈波輸出最小單位為 1/1,000,000 秒(uS)
- **State** 回傳變數值(0~255)，0:設定成功，255:設定失敗。

unsigned char **State** = GetTm3PulseOutState(void);

讀取 Timer 3 PulseOut 輸出狀態。

- **State** 回傳變數值(0~255)，回傳目前 PulseOut 輸出狀況，0: 輸出成功或無狀況，1: 等待 PulseOut 輸出完成，255: 讀取失敗。

```
#include "arminno.h"
int main(void)
{
  InitialGpioState(PD11, 2); // PD11 初為 High
  SetTm3PulseOut(15, 500, 0); // PD11, 輸出 500us low pulse
  while(GetTm3PulseOutState() == 1);
}
```

```
while(1);
}
```

unsigned char **State** = SetTm3PulseOutOff(void)

取消 Timer 3 PulseOut 功能設定。

- **State** 回傳變數值(0~255) · 0: 取消成功 · 255: 取消失敗。

頻率輸出功能

Timer 3 Frequency Out 功能可以輸出從 1Hz~1MHZ 之間的不同頻率之方波。

unsigned char **State** = SetTm3FreqOut(unsigned char **Output**,
unsigned long **Frequency**, unsigned char **Mode**);

設定 Timer 3 Frequency Out 功能並啟動。

- **Output** 常數或變數值(0~15) · 選擇不同輸出腳位 · 0:PA0 · 1:PA1 · 2:PA2 · 3:PA3 · 4:PA12 · 5:PA13 · 6:PA14 · 7:PA15 · 8:PC0 · 9:PC1 · 10:PC2 · 11:PC3 · 12:PD8 · 13:PD9 · 14:PD10 · 15:PD11 · 預設值為 0

Output	輸出腳位	Output	輸出腳位
0	PA0	8	PC0
1	PA1	9	PC1
2	PA2	10	PC2
3	PA3	11	PC3
4	PA12	12	PD8
5	PA13	13	PD9
6	PA14	14	PD10
7	PA15	15	PD11

- **Frequency** 常數或變數值(1~1,000,000) · 可選擇 1Hz~1MHz 的頻率輸出。預設值為 1
- **Mode** 常數或變數值(0~1) · 設定輸出狀態 · 0:push pull · 1:open drain · 預設值為 0
- **State** 回傳值(0~255) · 0:設定成功 · 255:設定失敗。

```
#include "arminno.h"
int main(void)
{
    SetTm3FreqOut(1, 2000); // PA1, 輸出 2kHz.
    while(1);
}
```

```
}

```

unsigned char **State** = SetTm3FreqOutOff(void);

取消 Timer 3 Frequency Out 輸出設定。

- **State** 回傳變數值(0~255) · 0: 取消成功 · 255: 取消失敗。

編碼器解碼功能

Timer 3 Decoder 功能可以讀取任何二線式編碼器(2-pin encoder)回授訊號。

unsigned char **State** = SetTm3Decoder(unsigned char **Input**,
unsigned short **Prescaler**, unsigned short **MaxCount**);

設定 Timer 3 Decoder 功能並啟動。

- **Input** 常數或變數值(0~3) · 選擇不同輸入腳位 · 0: PA0 和 PA1 · 1: PA12 和 PA13 · 2: PC0 和 PC1 · 3: PD8 和 PD9 · 預設值為 0

Input	輸入腳位
0	PA0 和 PA1
1	PA12 和 PA13
2	PC0 和 PC1
3	PD8 和 PD9

- **Prescaler** 常數或變數值(0~65535) · 可將輸入 Timer 3 之編碼器頻率除頻 · 設定 Timer 3 Decoder 可用單位(解析度)大小 · 公式如下: Timer 3 Decoder count 數值 = 外部編碼器輸入 count 數值 / (Prescaler + 1) · 預設值為 0
- **MaxCount** 常數或變數值(0~65535) · 選擇 Decoder count 最大值 · 超過最大值即產生溢位(overflow event) · 預設值為 65535
- **State** 回傳值(0~255) · 0: 設定成功 · 255: 設定失敗。

unsigned char **State** = SetTm3DecoderValue(short **TmValue**);

設定 Timer 3 Decoder 計數值。

- **TmValue** 常數和變數值(-32768~32767) · 設定 Timer 3 Decoder count 計數值。
- **State** 回傳變數值(0~255) · 0: 無狀況 · 255: 讀取失敗。

unsigned char **State** = GetTm3DecoderValue(short **TmValue**);

讀取 Timer 3 Decoder 計數值及狀態。

- **TmValue** 回傳變數值(-32768~32767) · 回傳 Timer 3 Decoder count 計數值。

- **State** 回傳變數值(0~255) · 0:無狀況 · 1:發生溢位(overflow or underflow) · 255:讀取失敗。

```
#include "arminno.h"
short TmValue;
int main(void)
{
    SetTm3Decoder (1); // PA12, PA13
    while(1) {
        GetTm3DecoderValue(TmValue); // 取得 decoder 的值
        printf("\r%d\r\n", TmValue);
        Pause(2000);
    }
}
```

unsigned char **State** = SetTm3DecoderOff(void);

取消 Timer 3 Decoder 設定。

- **State** 回傳變數值(0~255) · 0: 取消成功 · 255: 取消失敗。

Event 相關設定

開啟 Timer 3 Update Event 功能可於 Timer 3 計數數值產生溢位時(overflow or underflow)產生觸發事件(Event)，使用者於該觸發事件內撰寫相關程式。

unsigned char **State** = SetTm3Event(unsigned char **UpdateEvent**);

設定 Timer 3 UpdateEvent 功能。

- **UpdateEvent** 常數或變數值(0~1)，啟動或關閉 Update Event 功能 · 0: Disable · 1: Enable。
- **State** 回傳變數值(0~255) · 0:設定成功 · 255:設定失敗。

void Tm3UpdateEvent(void);

當發生 Tm3UpdateEvent 時(發生 overflow or underflow)，系統所設定之程式執行區塊。

Timer 4

Arminno™內建多功能硬體 Timer 4，特性如下：

- 16bit up/down auto reload counter register
- 內部/外部時間輸入計數功能，可產生固定時間事件(Event)
- 擁有 4 Channel 同時輸出之可程式化 PWM(Pulse Width Modulation)功能
- 可切換三組不同輸出入 I/O (請參考：[I/O 功能腳位對應表](#))
- 單一脈波輸入量測功能
- 單一脈波可調寬度輸出功能
- 二線式編碼器解碼功能
- Counter 數值溢位(underflow/overflow)事件觸發(Event trigger)功能

Timer 4 各函式使用功能介紹如下：

名稱	說明
產生固定時間 Event 和計數功能	
SetTm4Counter()	設定 Counter 相關參數並啟動
SetTm4CounterValue()	設定 Counter 計數初始值
GetTm4CounterValue()	讀取 Counter 計數狀態及數值
SetTm4CounterOff()	取消 Counter 設定
輸出 PWM 訊號功能	
SetTm4Pwm()	設定 PWM 相關參數並啟動
SetTm4PwmCh()	啟動及關閉 PWM channel 輸出
SetTm4PwmOff()	取消 PWM 設定
單一輸入脈波寬度量測功能	
SetTm4PulseIn()	設定 PulseIn 相關參數並啟動
GetTm4PulseInValue()	讀取 PulseIn 量測狀態及數值
SetTm4PulseInOff()	取消 PulseIn 設定
單一脈波輸出功能	
SetTm4PulseOut()	設定 PulseOut 相關參數並啟動
GetTm4PulseOutState()	讀取 PulseOut 輸出狀態
SetTm4PulseOutOff()	取消 PulseOut 設定
頻率輸出功能	
SetTm4FreqOut()	設定 FreqOut 相關參數並啟動
SetTm4FreqOutOff()	取消 FreqOut 設定
編碼器解碼功能	
SetTm4Decoder()	設定 Decoder 相關參數並啟動

SetTm4DecoderValue()	設定 Decoder 計數初始值
GetTm4DecoderValue()	讀取 Decoder 計數值
SetTm4DecoderOff()	取消 Decoder 設定
事件 (Event) 相關設定	
SetTm4Event()	設定 Timer overflow/underflow 觸發事件
Tm4UpdateEvent()	Timer overflow/underflow 觸發事件

Timer 相關功能設定，使用時在同一時間裡只能啟動其中一項功能，若要啟動不同的功能，則須先將之前功能設定取消後再啟動另一功能，詳細說明如下：

產生固定時間 Event 和計數功能

使用 Timer 4 counter 功能除了可利用內外部時鐘輸入來設定自動計數功能之外，更可讓 Arminno™於程式執行期間，於固定時間間隔(overflow or underflow)產生狀態通知(state)或事件(Event)，來提醒使用者於固定時間處理特定事情。

```
unsigned char State = SetTm4Counter( unsigned char InputMode,
                                     unsigned char CountMode,
                                     unsigned short Prescaler,
                                     unsigned short MaxCount);
```

設定 Timer 4 counter 功能並啟動。

- **InputMode** 常數或變數值(0~2)，選擇 Counter clock 來源。0: 內部系統時鐘 Fclk(72MHz)，1: 外部時鐘輸入(PC10)，2: 外部時鐘輸入(PE4)，3:外部時鐘輸入(PE10)。預設值為 0

<i>InputMode</i>	時鐘輸入源
0	內部系統時鐘 Fclk(72MHz)
1	外部時鐘輸入(PC10)
2	外部時鐘輸入(PE4)

- **CountMode** 常數或變數值(0~2)，Counter 計數方式，0:遞增(up)，1: 遞減(down)，2: 遞增+遞減循環(center)。預設值為 0
- **Prescaler** 常數或變數值(0~65535)，可將輸入之時鐘頻率除頻，進而調整計數速度，公式為： 頻率=內部(外部)時鐘輸入/(Prescaler+1)。預設值為 719，若為 72MHz 內部時鐘輸入時，除頻之輸入時鐘單位為 1/100,000 秒
- **MaxCount** 常數或變數值(0~65535)，設定此數值可限制 Counter 最大值。預設值為 10000，配合預設 Prescaler 則為每 0.1 秒產生狀態通知(state)或事件(Event)
- **State** 回傳變數值(0~255)，0: 設定成功，255: 設定失敗。

unsigned char ***State*** = SetTm4CounterValue(unsigned short ***TmValue***);

設定 Timer 4 counter 之數值，通常於設定初始值時使用。

- ***TmValue*** 常數或變數值(0~65535)，為直接設定 Counter 數值。
- ***State*** 回傳變數值(0~255)，0: 設定成功，255: 設定失敗。

unsigned char ***State*** = GetTm4CounterValue(unsigned short ***TmValue***);

讀取 Timer 4 counter 之數值。

- ***TmValue*** 回傳變數值(0~65535)，讀取 Counter 數值。
- ***State*** 回傳變數值(0~255)，0: 無狀況，1: 產生 updated Event(overflow or underflow)，255: 讀取失敗。

```
#include "arminno.h"
#define Led2      PA15
unsigned long TmValue;
void Tm4UpdateEvent(void)
{
    Toggle(Led2);
}
int main(void)
{
    InitialGpioState(Led2, 1);
    SetTm4Counter (0, 0, 719, 25000); // 設定 TM4 週期為 0.25 秒
    SetTm4Event(1);
    while(1);
}
```

unsigned char ***State*** = SetTm4CounterOff(void);

取消 Timer 4 counter 之功能設定。

- ***State*** 回傳變數值(0~255)，0: 取消成功，255: 取消失敗。

輸出 PWM 訊號功能

使用 Timer 4 PWM 的功能除了可以透過自訂 PWM 頻率及 PWM 解析度功能來達成所需要的 PWM 波形之外，更可藉由最多四個 Channel 輸出的功能來達到多元件同步控制的效果。

unsigned char ***State*** = SetTm4Pwm(unsigned short ***Prescaler***,
unsigned short ***Resolution***);

設定 Timer 4 PWM 功能並啟動。

- **Prescaler** 常數或變數值(0~65535)，可將輸入之時鐘頻率做除頻動作，進而調整 PWM 每單位時間速度，公式如下：速度=72MHz/(Prescaler+1)。預設值為 71，若為 72MHz 內部時鐘輸入時，除頻之輸入時鐘單位為 1/1,000,000 秒
- **Resolution** 常數或變數值(0~65535)，為直接設定 PWM 可用解析度範圍數值。預設值為 1000，配合預設 Prescaler 則為產生解析度為 1000 單位之 1KHz PWM 波型輸出
- **State** 回傳變數值(0~255)，0:設定成功，255:設定失敗。

```
unsigned char State = SetTm4PwmCh( unsigned char Channel,
                                   unsigned char Enable, unsigned char Mode,
                                   unsigned short Duty);
```

設定 Timer 4 PWM 所要輸出之 Channel 設定。

- **Channel** 常數或變數值(0~3)，選擇 PWM Channel。Channel 選擇請參考下面表列。
- **Enable** 常數或變數值(0~3)，0: 關閉輸出，1:開啟輸出(Ch0:PD12、Ch1:PD14、Ch2:PE0 或 Ch3:PE2)，2:開起輸出(Ch0:PC13、Ch1:PC11、Ch2:PD8 或 Ch3:PD10)，3:開起輸出(Ch0:PE14)。預設值為 0

Enable	Ch0	Ch1	CH2	Ch3
0	關閉輸出	關閉輸出	關閉輸出	關閉輸出
1	PD12	PD14	PE0	PE2
2	PC13	PC11	PD8	PD10
3	PE14	-	-	-

- **Mode** 常數或變數值(0~1)，0:push pull，1:open drain。預設值為 0
- **Duty** 常數或變數值(0~65535)，設定 PWM Channel 之比例輸出值。預設值為 0
- **State** 回傳變數值(0~255)，0: 設定成功，255: 設定失敗。

```
#include "arminno.h"
int main(void)
{
    SetTm4Pwm ();           // 週期 1ms
    while(1) {
        SetTm4PwmCh(0, 1, 0, 100); // PD12, duty = 100us
        SetTm4PwmCh(1, 1, 0, 200); // PD14, duty = 200us
        SetTm4PwmCh(2, 1, 0, 300); // PE0, duty = 300us
    }
}
```

```
SetTm4PwmCh(3, 1, 0, 400);    // PE2, duty = 400us
Pause(20000);                  // 延遲 2 秒
SetTm4PwmCh(0, 1, 0, 900);    // PD12, duty = 900us
SetTm4PwmCh(1, 1, 0, 800);    // PD14, duty = 800us
SetTm4PwmCh(2, 1, 0, 700);    // PE0, duty = 700us
SetTm4PwmCh(3, 1, 0, 600);    // PE2, duty = 600us
Pause(20000);                  // 延遲 2 秒
}
}
```

unsigned char **State** = SetTm4PwmOff(void)

- 取消 Timer 4 PWM 之功能設定。
- **State** 回傳變數值(0~255) · 0: 取消成功 · 255: 取消失敗。

單一輸入脈波寬度量測功能

Timer 4 Pulse In 功能可以完全精準的量測所要輸入的方波波形寬度，並可自訂量測刻度大小。

unsigned char **State** = SetTm4PulseIn(unsigned char **Input**,
unsigned char **Trigger**, unsigned short **Prescaler**);

- 設定 Timer 4 Pulse In 功能並啟動。
- **Input** 常數或變數值(0~4)，選擇不同輸入腳位 · 0:PD12 · 1:PD14 · 2:PC13 · 3:PC11 · 4:PE14。預設值為 0

<i>Input</i>	輸入腳位	<i>Input</i>	輸入腳位
0	PD12	3	PC11
1	PD14	4	PE14
2	PC13		

- **Trigger** 常數或變數值(0~1)，所要量測 Pulse 類型 · 0:Low pulse 及 1:High pulse。預設值為 0
- **Prescaler** 常數或變數值(0~65535)，可將輸入 Timer1 之時鐘頻率除頻，進而設定 PulseIn 寬度量測可用單位大小，公式如下：PulseIn 量測頻率 = 72MHz/(Prescaler+1)。預設值為 71，依照預設值設定則脈波輸出最小單位為 1/1,000,000 秒(uS)
- **State** 回傳變數值(0~255) · 0:設定成功 · 255:設定失敗。

unsigned char **State** = GetTm4PulseInValue(unsigned Short **Value**);

讀取 Timer 4 PulseIn 量測數值和狀態。

- **Value** 回傳變數值(0~65535) · PulseIn 量測數值回傳。
- **State** 回傳值(0~255) · 回傳目前 PulseIn 量測狀況 · 0: 設定成功 · 1: 等待 PulseIn 量測完成 · 2: 超過所能量測最大值(overflow) · 255: 讀取失敗。

```
#include "arminno.h"
int main(void)
{
    unsigned short Value;
    unsigned char status;
    SetTm4PulseIn (4);    // PE14
    do {
        status = GetTm4PulseInValue(Value);
    } while(status == 1);
    if(status == 0) {
        printf("Pulse In = %d\r\n", Value); // 成功
    } else if(status == 2) {
        printf("Overflow\r\n");           // 溢位
    } else {
        printf("Error\r\n");              // 失敗
    }
}
```

unsigned char **State** = SetTm4PulseInOff(void)

取消 Timer 4 PulseIn 功能設定。

- **State** 回傳變數值(0~255) · 0: 取消成功 · 255: 取消失敗。

單一脈波輸出功能

Timer 4 Pulse Out 功能可針對不同使用設定 · 輸出一設定寬度之方波訊號

unsigned char **State** = SetTm4PulseOut(unsigned char **Output**,
unsigned char **Width**, unsigned short **Level**,
unsigned short **Prescaler**);

設定 Timer 4 Pulse Out 功能並啟動。

- **Output** 常數或變數值(0~8) · 選擇不同輸出腳位 · 0:PD12 · 1:PD14 · 2:PE0 · 3:PE2 · 4:PC13 · 5:PC11 · 6:PD8 · 7:PD10 · 8:PE14 · 預設值為 0

<i>Input</i>	輸入腳位	<i>Input</i>	輸入腳位
0	PD12	5	PC11
1	PD14	6	PD8
2	PE0	7	PD10
3	PE2	8	PE14
4	PC13		

- **Width** 常數或變數值(0~65535)，選擇要輸出 Pulse 寬度。預設值為 1000
- **Level** 常數或變數值(0~1)，所要輸出 Pulse 類型。0: Low pulse 及 1: High pulse。預設值為 1
- **Prescaler** 常數或變數值(0~65535)，將輸入 Timer 4 之時鐘頻率除頻，設定 PulseOut 寬度輸出可用單位大小，公式如下：PulseOut 輸出單位頻率 = 72MHz/(Prescaler+1)。預設值為 71，依照預設值設定則脈波輸出最小單位為 1/1,000,000 秒(us)
- **State** 回傳變數值(0~255)，0:設定成功，255:設定失敗。

unsigned char **State** = GetTm4PulseOutState(void);

讀取 Timer 4 PulseOut 輸出狀態。

- **State** 回傳變數值(0~255)，回傳目前 PulseOut 輸出狀況。0: 輸出成功或無狀況，1: 等待 PulseOut 輸出完成，255: 讀取失敗。

unsigned char **State** = SetTm4PulseOutOff(void)

取消 Timer 4 PulseOut 功能設定。

- **State** 回傳變數值(0~255)，0: 取消成功，255: 取消失敗。

```
#include "arminno.h"
int main(void)
{
    InitialGpioState(PD14, 1); // PD14, 輸出 LOW
    SetTm4PulseOut(1, 512); // PD14, 512us high pulse
    while(GetTm4PulseOutState() == 1);
    return 0;
}
```

頻率輸出功能

Timer 4 Frequency Out 功能可以輸出從 1Hz~1MHZ 之間的不同頻率之方波。

unsigned char **State** = SetTm4FreqOut(unsigned char **Output**,
unsigned long **Frequency**, unsigned char **Mode**);

設定 Timer 4 Frequency Out 功能並啟動。

- **Output** 常數或變數值(0~8)，選擇不同輸出腳位，0:PD12，1:PD14，2:PE0，3:PE2，4:PC13，5:PC11，6:PD8，7:PD10，8:PE14。預設值為 0

<i>Input</i>	輸入腳位	<i>Input</i>	輸入腳位
0	PD12	5	PC11
1	PD14	6	PD8
2	PE0	7	PD10
3	PE2	8	PE14
4	PC13		

- **Frequency** 常數或變數值(1~1,000,000)，可選擇 1Hz~1MHz 的頻率輸出。預設值為 1
- **Mode** 常數或變數值(0~1)，設定輸出狀態，0:push pull，1:open drain。預設值為 0
- **State** 回傳值(0~255)，0:設定成功，255:設定失敗。

```
#include "arminno.h"
int main(void)
{
    SetTm4FreqOut(4, 1); // PC13, 1Hz
    while(1);
}
```

unsigned char **State** = SetTm4FreqOutOff(void);

取消 Timer 4 Frequency Out 輸出設定。

- **State** 回傳變數值(0~255)，0: 取消成功，255: 取消失敗。

編碼器解碼功能

Timer 4 Decoder 功能可以讀取任何二線式編碼器(2-pin encoder)回授訊號。

unsigned char **State** = SetTm4Decoder(unsigned char **Input**,
unsigned short **Prescaler**, unsigned short **MaxCount**);

設定 Timer 4 Decoder 功能並啟動。

- **Input** 常數或變數值(0~3)，選擇不同輸入腳位，0:PD12 和 PD14，1:PC13 和

PC11。預設值為 0

<i>Input</i>	輸入腳位
0	PD12 和 PD14
1	PC13 和 PC11

- **Prescaler** 常數或變數值(0~65535)·可將輸入 Timer 4 之編碼器頻率除頻·設定 Timer 4 Decoder 可用單位(解析度)大小·公式如下: Timer 4 Decoder count 數值=外部編碼器輸入 count 數值/(Prescaler+1)·預設值為 0
- **MaxCount** 常數或變數值(0~65535)·選擇 Decoder count 最大值·超過最大值即產生溢位(overflow event)·預設值為 65535
- **State** 回傳值(0~255)·0:設定成功·255:設定失敗。

unsigned char **State** = **SetTm4DecoderValue**(short *TmValue*);

設定 Timer 4 Decoder 計數值。

- **TmValue** 常數和變數值(-32768~32767)·設定 Timer 4 Decoder count 計數值。
- **State** 回傳變數值(0~255)·0:無狀況·255:讀取失敗。

unsigned char **State** = **GetTm4DecoderValue**(short *TmValue*);

讀取 Timer 3 Decoder 計數值及狀態。

- **TmValue** 回傳變數值(-32768~32767)·回傳 Timer 4 Decoder count 計數值。
- **State** 回傳變數值(0~255)·0:無狀況·1:發生溢位(overflow or underflow)·255:讀取失敗。

```
#include "arminno.h"
short TmValue;
int main(void)
{
    SetTm4Decoder (); // PD12, PD14
    while(1) {
        GetTm4DecoderValue(TmValue);
        printf("\r%d\r\n", TmValue);
        Pause(2000);
    }
}
```

unsigned char ***State*** = SetTm4DecoderOff(void);

取消 Timer 4 Decoder 設定。

- ***State*** 回傳變數值(0~255) , 0: 取消成功 , 255: 取消失敗。

Event 相關設定

開啟 Timer 4 Update Event 功能可於 Timer 4 計數數值產生溢位時(overflow or underflow)產生觸發事件(Event) , 使用者於該觸發事件內撰寫相關程式。

unsigned char ***State*** = SetTm4Event(unsigned char ***UpdateEvent***);

設定 Timer 4 UpdateEvent 功能。

- ***UpdateEvent*** 常數或變數值(0~1) , 啟動或關閉 Update Event 功能 , 0: Disable、1: Enable。
- ***State*** 回傳變數值(0~255) , 0:設定成功 , 255:設定失敗。

void Tm4UpdateEvent(void);

當發生 Tm4UpdateEvent 時(發生 overflow or underflow) , 系統所設定之程式執行區塊。

RTC

RTC(Real Time Clock)為一個擁有慢速時鐘來源輸入(32,768Hz)，並可像時鐘一樣輸出實際時間的 Timer。RTC Timer 擁有比一般用途 Timer 更為精確的 Timer 數值溢位(overflow)調整範圍，除了可輸出實際時間之外，也可拿來當做一般 Timer 運用。Arminno™內建多功能硬體 RTC 函式庫功能，其具備各種 Timer 相關功能，其特性如下：

- 32bit up auto reload counter register
- 內部時間輸入計數功能，可產生固定時間事件(Event)
- Counter 數值溢位(underflow/overflow)事件觸發(Event trigger)功能

RTC 函式庫各函式使用功能介紹如下：

名稱	說明
SetRtc()	設定 RTC 相關參數並啟動
GetRtcValue()	讀取 RTC 計數狀態及數值
GetRtcClock()	讀取 RTC 每次計數時間狀態
SetRtcOff()	取消 RTC 設定
事件 (Event) 相關設定	
SetRtcEvent()	設定 RTC 觸發事件
RtcMatchEvent()	RTC 數值符合事件
RtcClkEvent()	RTC 每次計數時間事件

指令詳細說明

unsigned char **State** = SetRtc(unsigned char **Prescaler**, unsigned long **Match**);

設定 RTC 相關設定並啟動。

- **Prescaler** 常數或變數值(0~15)，選擇 RTC 時鐘來源(32,768Hz)的除頻數值，數值範圍為 $2^{0 \sim \text{Prescaler}}$ 公式為: $\text{RTC 時鐘來源} = 32768\text{Hz} / 2^{\text{Prescaler}}$ 。預設值為 0
- **Match** 常數或變數值(0~4294967295)，設定 RTC Counter 觸發符合數值，設定之後若是 RTC counter value 遞增到符合此數值(發生溢位 overflow)，則啟動觸發狀態通知或事件(Event)通知。預設值為 32768
- **State** 回傳變數值(0~255)，0: 無狀況，255: 設定失敗。

unsigned char **State** = GetRtcValue(unsigned long **Value**);

讀取 RTC counter 數值及狀態。

- **Value** 讀取 RTC Counter 數值(0~4294967295)。

- **State** 回傳變數值(0~255) · 0: 無狀況 · 1: 溢位(overflow) · 255:讀取失敗。

```
#include "arminno.h"
unsigned long Value;
int main(void)
{
    SetRtc(8, 125);
    while(1) {
        GetRtcValue(Value);
        printf("%d\r\n", Value);
        Pause(10000);
    }
}
```

unsigned char **State** = **GetRtcClk**(void);

讀取 RTC 時鐘來源狀態(除頻後)。

- **State** 回傳值(0~255) · 0:無單位時間觸發 · 1:發生單位時間觸發 · 255:讀取失敗。

unsigned char **State** = **SetRtcOff**(void);

取消 RTC 設定。

- **State** 回傳變數值(0~255) · 0:取消成功 · 255:取消失敗。

Event 相關設定

unsigned char **State** = **SetRtcEvent**(unsigned char **Match**, unsigned char **Clock**);

設定 RTC 事件(Event)觸發功能。

- **Match** 常數或變數值(0~1) · 啟動或關閉數值符合事件(RtcMatchEvent) · 0:Disable · 1:Enable。
- **Clock** 常數或變數值(0~1) · 啟動或關閉時鐘來源觸發事件(RtcClkEvent) · 0:Disable · 1:Enable。
- **State** 回傳變數值(0~255) · 0:無狀況 · 255:設定失敗。

void **RtcMatchEvent**(void);

當發生 RtcMatchEvent 時(數值符合事件、發生 overflow) · 系統所設定之程式執行區塊。

`void RtcClkEvent(void);`

當發生 RtcClkEvent(時鐘來源觸發事件)時，系統所設定之程式執行區塊。

```
#include "arminno.h"
unsigned int count = 0;
unsigned int x = 0;
void RtcClkEvent(void)
{
    if(++x >= 128) {
        x = 0;
        printf("%d\r\n", count++);
    }
}
int main(void)
{
    SetRtc(8, 125);
    SetRtcEvent(0, 1);
    while(1);
}
```

OPA0/CMP0

Arminno™內建運算放大器 OPA0(Operational amplifier, OPA)及類比比較器 CMP0(Analog Comparator, CMP)功能。只需切換模式即可作為運算放大器或是類比比較器使用。OPA0/CMP0 函式庫功能介紹如下：

名稱	說明
SetCmp0()	設定 Comparator 0 相關參數並啟動
GetCmp0()	讀取 Comparator 0 觸發狀態
SetCmp0Off()	取消 Comparator 0 設定
事件 (Event) 相關設定	
SetCmp0Event()	設定 Comparator 0 觸發事件
Cmp0FallingEdgeEvent()	Comparator 0 下降緣觸發事件
Cmp0RisingEdgeEvent()	Comparator 0 上升緣觸發事件

指令詳細說明

unsigned char **State** = SetCmp0(unsigned char **Mode**);

設定 OPA0/CMP0 操作模式並啟動。輸入腳位 PE5:CN0、PE6:CP0，輸出腳位 PE7:Aout0。(請參考：[I/O 功能腳位對應表](#))

- **Mode** 常數或變數值(0~1)，選擇操作模式，0: 運算放大器(Operational Amplifier)·1: 比較器(Comparator)無 PE7:AOUT 輸出·2: 比較器(Comparator)有 PE7:AOUT 輸出。預設值為 0
- **State** 回傳變數值(0~255)，0: 無狀況，255: 設定失敗。

unsigned char **State** = GetCmp0(void);

讀取 CMP0 比較結果。

- **State** 回傳變數值(0~255)，0: 無狀況，1: 下降緣觸發(Falling edge trigger)，2: 上升緣觸發(Rising edge trigger)，3: 下降緣觸發和上升緣觸發同時存在，255:讀取失敗。

unsigned char **State** = SetCmp0Off(void);

取消 OPA0/CMP0 設定。

- **State** 回傳變數值(0~255)，0: 取消成功，255: 取消失敗。

Event 相關設定

unsigned char **State** = SetCmp0Event(unsigned char **Falling**,
unsigned char **Rising**);

設定 OPA0/CMP0 事件(Event)觸發功能。

- **Falling** 常數或變數值(0~1) · 啟動或關閉下降緣觸發事件(FallingEdgeEvent) · 0: Disable · 1: Enable。
- **Rising** 常數或變數值(0~1) · 啟動或關閉上升緣觸發事件(RisingEdgeEvent) · 0: Disable · 1: Enable。
- **State** 回傳變數值(0~255) · 0: 無狀況 · 255: 設定失敗。

void Cmp0FallingEdgeEvent(void);

當發生下降緣觸發事件(FallingEdgeEvent)時，系統所設定之程式執行區塊。

void Cmp0RisingEdgeEvent(void);

當發生上升緣觸發事件(RisingEdgeEvent)時，系統所設定之程式執行區塊。

```
#include "arminno.h"
void Cmp0FallingEdgeEvent(void)
{
    printf("Falling\r\n");
}
void Cmp0RisingEdgeEvent(void)
{
    printf("Rising\r\n");
}
int main(void)
{
    SetCmp0 (1);
    SetCmp0Event (1, 1);
    while(1);
}
```

OPA1/CMP1

Arminno™內建運算放大器 OPA1(Operational amplifier, OPA)及類比比較器 CMP1(Analog Comparator, CMP)功能。只需切換模式即可作為運算放大器或是類比比較器使用。OPA1/CMP1 函式庫功能介紹如下：

名稱	說明
SetCmp1()	設定 Comparator 1 相關參數並啟動
GetCmp1()	讀取 Comparator 1 觸發狀態
SetCmp1Off()	取消 Comparator 1 設定
事件 (Event) 相關設定	
SetCmp1Event()	設定 Comparator 1 觸發事件
Cmp1FallingEdgeEvent()	Comparator 1 下降緣觸發事件
Cmp1RisingEdgeEvent()	Comparator 1 上升緣觸發事件

指令詳細說明

unsigned char **State** = SetCmp1(unsigned char **Mode**);

設定 OPA1/CMP1 操作模式並啟動。輸入腳位 PE8:CN0、PE9:CP0，輸出腳位 PE10:Aout1。(請參考：[I/O 功能腳位對應表](#))

- **Mode** 常數或變數值(0~1)，選擇操作模式，0: 運算放大器(Operational Amplifier)，1: 比較器(Comparator)無 PE10:Aout1 輸出，2: 比較器(Comparator)有 PE10:Aout1 輸出。預設值為 0
- **State** 回傳變數值(0~255)，0: 無狀況，255: 設定失敗。

unsigned char **State** = GetCmp1(void);

讀取 CMP1 比較結果。

- **State** 回傳變數值(0~255)，0: 無狀況，1: 下降緣觸發(Falling edge trigger)，2: 上升緣觸發(Rising edge trigger)，3: 下降緣觸發和上升緣觸發同時存在，255:讀取失敗。

unsigned char **State** = SetCmp1Off(void);

取消 OPA1/CMP1 設定。

- **State** 回傳變數值(0~255)，0: 取消成功，255: 取消失敗。

Event 相關設定

unsigned char **State** = SetCmp1Event(unsigned char **Falling**,
unsigned char **Rising**);

設定 OPA1/CMP1 事件(Event)觸發功能。

- **Falling** 常數或變數值(0~1) · 啟動或關閉下降緣觸發事件(FallingEdgeEvent) · 0: Disable · 1: Enable。
- **Rising** 常數或變數值(0~1) · 啟動或關閉上升緣觸發事件(RisingEdgeEvent) · 0: Disable · 1: Enable。
- **State** 回傳變數值(0~255) · 0: 無狀況 · 255: 設定失敗。

void Cmp1FallingEdgeEvent(void);

當發生下降緣觸發事件(FallingEdgeEvent)時，系統所設定之程式執行區塊。

void Cmp1RisingEdgeEvent(void);

當發生上升緣觸發事件(RisingEdgeEvent)時，系統所設定之程式執行區塊。

```
#include "arminno.h"
void Cmp1FallingEdgeEvent(void)
{
    printf("Falling\r\n");
}
void Cmp1RisingEdgeEvent(void)
{
    printf("Rising\r\n");
}
int main(void)
{
    SetCmp1 (1);
    SetCmp1Event (1, 1);
    while(1);
}
```

EE-DATA

Arminno™利用演算法將內部 Flash 記憶體模擬 EEPROM 功能，每個單獨之 EEPROM Address(0~4095)擁有 2bytes 儲存空間，使用者可以針對個別單獨之 EEPROM Address 進行讀寫使用，函式庫功能介紹如下：

名稱	說明
WriteEEData()	將資料寫入 EEPROM
ReadEEData()	從 EEPROM 讀取資料

指令詳細說明

unsigned char *State* = WriteEEData(unsigned short *Address*, void **Variable*,
unsigned short *Length*);

將資料寫入至 EEPROM 記憶體中。

- *Address* 常數或變數值(0~4095)，指定 EEPROM 位址。
- *Variable* 指標變數值，傳遞寫入之變數記憶體位址。
- *Length* 常數或變數值(1~65535)，傳遞所要儲存變數之長度，若是超出該位址空間大小則會剩餘資料依序存入該位址+1、該位址+2....之儲存空間，直到全部長度資料儲存完畢。
- *State* 回傳變數值(0~255)，0: 無狀況，255: 寫入失敗。

unsigned char *State* = ReadEEData(unsigned short *Address*, void **Variable*,
unsigned short *Length*);

從 EEPROM 記憶體中讀取資料。

- *Address* 常數或變數值(0~4095)，設定所要讀取資料之 EEPROM 位址。
- *Variable* 指標變數，傳遞讀取之變數記憶體位址。
- *Length* 常數或變數值(1~65535)，傳遞所要讀取之變數長度，若是超出該位址空間大小則會依序取出該位址+1、該位址+2....之儲存空間資料，直到全部長度資料讀取完成。
- *State* 回傳變數值(0~255)，0: 無狀況，255: 讀取失敗。

```
#include "arminno.h"
unsigned short Data;
unsigned short i;
int main(void) {
    for(i = 0 ; i < 128 ; i++)
        WriteEEData(i, &i, 2);
    for(i = 0 ; i < 128 ; i++) {
        ReadEEData(i, &Data, 2);
        if(Data != i) {
            printf("FAIL\r\n");
            while(1);
        }
    }
    printf("OK\r\n");
    while(1);
}
```

I2C0

Arminno™所提供之 I2C 為一串列通訊匯流排，使用主從架構(Master and Slave)，於 1980 年代由飛利浦公司發展而來，其使用 2 pin 實體資料傳輸線(SDA、SCL)，並採用 open drain + 電阻上拉模式交換串列資訊，可進行一對一及一對多模組資料傳輸，Arminno™上允許 I2C 工作於 3.3V 及 5V、主從模式切換以及 100kbps、400kbps 及 1Mbps 的傳輸速度模式設定。函式庫功能介紹如下：

名稱	說明
SetI2c0()	設定相關參數並啟動
I2c0MasterTransmit()	Master 資料輸出
I2c0MasterReceive()	Master 資料輸入
GetI2c0MasterState()	讀取 Master 狀態
GetI2c0SlaveState()	讀取 Slave 狀態
I2c0SlaveTransmit()	Slave 資料輸出
I2c0SlaveReceive()	Slave 資料輸入
SetI2c0Off()	取消設定
I2C 事件 (Event) 設定	
SetI2c0Event()	設定 Event 功能
I2c0MasterEvent()	Master 狀態 Event
I2c0SlaveEvent()	Slave 狀態 Event

指令詳細說明

```
unsigned char State = SetI2c0(unsigned char Speed,
                                unsigned char Pin,
                                unsigned short Address,
                                unsigned char GeneralCallEnable,
                                unsigned char AddressMode);
```

設定 I2C0 操作模式並啟動(請參考：[I/O 功能腳位對應表](#))。

- *Speed* 常數或變數值(0~2)，設定運行速度，0: 100kbps，1: 400kbps，2: 1MHzbps。預設值為 0
- *Pin* 常數或變數值(0~2)，設定 SCL 及 SDA 腳位，0: PC4 及 PC5，1: PC11 及 PC12，2: PD12 及 PD13。預設值為 0

<i>Pin</i>	SCL 及 SDA
0	PC4 及 PC5
1	PC11 及 PC12

- **Address** 常數或變數值(0~1023) · 設定在 Slave 模式時之 Slave ID · 7bit Address: 0~127 · 10bit Address: 0~1023 · 預設值為 1
- **GeneralCallEnable** 常數或變數值(0~1) · 設定 General Call 啟動或取消(Slave ID=0) · 0: Disable · 1: Enable · 預設值為 0
- **AddressMode** 常數或變數值(0~1) · 設定 7bit 或 10bit Address 模式 · 0: 7bit Address · 1: 10bit Address · 預設值為 0
- **State** 回傳變數值(0~255) · 0: 無狀況 · 255: 設定失敗。

```
unsigned char State = I2C0MasterTransmit( void *Array,
                                         unsigned short Length,
                                         unsigned short Address,
                                         unsigned char AddressMode);
```

I2C0 Master 模式資料輸出功能。

- **Array** 指標變數 · 傳遞所要輸出之變數記憶體位址。
- **Length** 常數或變數值(0~65535) · 傳遞所要輸出之變數長度 · 設定 0 或 1 皆為長度 1。
- **Address** 常數或變數值(0~1023) · 設定 Target I2C Slave ID · 7bit Address: 0~127 · 10bit Address: 0~1023。
- **AddressMode** 常數或變數值(0~1) · 設定 7bit 或 10bit Address 模式 · 0: 7bit Address · 1: 10bit Address · 預設值為 0
- **State** 回傳值(0~255) · 0: 無狀況 · 255: 設定失敗。

```
unsigned char State = I2C0MasterReceive(void *Array,
                                         unsigned short Length,
                                         unsigned short Address,
                                         unsigned char AddressMode);
```

I2C0 Master 模式資料輸入功能。

- **Array** 指標變數 · 傳遞所要輸入之變數記憶體位址。
- **Length** 常數或變數值(0~65535) · 傳遞所要輸入之變數長度 · 設定 0 或 1 皆為長度 1 · 預設值為 0
- **Address** 常數或變數值(0~1023) · 設定 Target I2C Slave ID · 7bit Address: 0~127 · 10bit Address: 0~1023。
- **AddressMode** 常數或變數值(0~1) · 設定 7bit 或 10bit Address 模式 · 0: 7bit Address · 1: 10bit Address · 預設值為 0
- **State** 回傳值(0~255) · 0: 無狀況 · 255: 設定失敗。

unsigned char **State** = GetI2c0MasterState(void);

讀取 I2C0 Master 模式狀態通知。

- **State** 回傳值(0~255)：0: 無狀況，1: 資料傳輸完成，2: 資料傳輸失敗，3: 資料傳輸中，255: 讀取狀態錯誤。

unsigned char **State** = GetI2c0SlaveState(void);

讀取 I2C0 Slave 模式狀態通知。

- **State** 回傳值(0~255)：0: 無狀況，1: 資料傳輸完成，2: 資料傳輸失敗(包括資料長度不符)，3: 資料傳輸中，4: Slave 模式準備資料輸出，5: Slave 模式準備資料輸入，6: 準備 General Call 資料輸出，7: 準備 General Call 資料輸入，255: 讀取狀態錯誤。

unsigned char **State** = I2c0SlaveTransmit(void ***Array**, unsigned short **Length**);

I2C0 Slave 模式資料輸出功能。

- **Array** 指標變數，傳遞所要輸出之變數記憶體位址。
- **Length** 常數或變數值(0~65535)，傳遞所要輸出之變數長度，設定 0 或 1 皆為長度 1。預設值為 0
- **State** 回傳值(0~255)：0: 無狀況，255: 設定失敗。

unsigned char **State** = I2c0SlaveReceive(void ***Array**, unsigned short **Length**);

I2C0 Slave 模式資料輸入功能。

- **Array** 指標變數，傳遞所要入之變數記憶體位址
- **Length** 常數或變數值(0~65535)，傳遞所要輸入之變數長度，設定 0 或 1 皆為長度 1。預設值為 0
- **State** 回傳值(0~255)：0: 無狀況，255: 設定失敗。

```
#include "arminno.h"
#define EEAddr 0x51
unsigned char DataBuf[12];
int main(void) {
    SetI2c0(0, 1, 0, 0);
    // Write EEPROM
    DataBuf[0] = 0;
    DataBuf[1] = 0xA0;
    DataBuf[2] = 0x23;
    DataBuf[3] = 0x11;
    I2c0MasterTransmit(DataBuf, 4, EEAddr, 0);
```

```

while(GetI2c0MasterState() == 3);

    Pause(100);
// Read EEPROM
    DataBuf[0] = 0;
    DataBuf[1] = 0;
    I2c0MasterTransmit(DataBuf, 1, EEAddr, 0);
    while(GetI2c0MasterState() == 3);

    Pause(10);

    DataBuf[0] = DataBuf[1] = 0xAA;
    while(I2c0MasterReceive(DataBuf, 3, EEAddr, 0));
    while(GetI2c0MasterState() == 3);

    if((DataBuf[0] != 0xA0) || (DataBuf[1] != 0x23) ||
        (DataBuf[2] != 0x11))
        printf("Error\r\n");
    while(1);
}

```

unsigned char **State** = SetI2c0Off(void);

取消 I2C0 功能設定。

- **State** 回傳值(0~255) · 0:取消成功 · 255:取消失敗。

Event 相關設定

unsigned char **State** = SetI2c0Event(unsigned char **MasterEventEnable**,
unsigned char **SlaveEventEnable**);

設定 I2C 事件(Event)觸發功能。

- **MasterEventEnable** 常數或變數值(0~1) · 啟動或關閉 I2C0 Master 模式事件 · 0: Disable · 1: Enable。
- **SlaveEventEnable** 常數或變數值(0~1) · 啟動或關閉 I2C0 Slave 模式事件 · 0: Disable · 1: Enable。
- **State** 回傳值(0~255) · 0: 無狀況 · 255: 設定失敗。

void I2c0MasterEvent(void);

當 GetI2c0MasterState()回傳值轉變為 2(資料傳輸完成)或 3(資料傳輸失敗)時，系統所設定之程式執行區塊。

<i>state</i>	狀態說明
1	資料傳輸完成
2	資料傳輸失敗

void I2c0SlaveEvent(void);

當 GetI2c0SlaveState()回傳值轉變為 2(資料傳輸完成)、3(資料傳輸失敗)、4(Slave 模式準備資料輸出)、5(Slave 模式準備資料輸入)、6(準備 General Call 資料輸出)或 7(準備 General Call 資料輸入)時，系統所設定之程式執行區塊。

<i>state</i>	狀態說明
1	資料傳輸完成
2	資料傳輸失敗
4	Slave 模式準備資料輸出
5	Slave 模式準備資料輸入
6	準備 General Call 資料輸出
7	準備 General Call 資料輸入

I2C1

Arminno™所提供之 I2C 為一串列通訊匯流排，使用主從架構(Master and Slave)，於 1980 年代由飛利浦公司發展而來，其使用 2 pin 實體資料傳輸線(SDA、SCL)，並採用 open drain + 電阻上拉模式交換串列資訊，可進行一對一及一對多模組資料傳輸，Arminno™上允許 I2C 工作於 3.3V 及 5V、主從模式切換以及 100kbps、400kbps 及 1Mbps 的傳輸速度模式設定。函式庫功能介紹如下：

名稱	說明
SetI2c1()	設定相關參數並啟動
I2c1MasterTransmit()	Master 資料輸出
I2c1MasterReceive()	Master 資料輸入
GetI2c1MasterState()	讀取 Master 狀態
GetI2c1SlaveState()	讀取 Slave 狀態
I2c1SlaveTransmit()	Slave 資料輸出
I2c1SlaveReceive()	Slave 資料輸入
SetI2c1Off()	取消設定
I2C 事件 (Event) 設定	
SetI2c1Event()	設定 Event 功能
I2c1MasterEvent()	Master 狀態 Event
I2c1SlaveEvent()	Slave 狀態 Event

指令詳細說明

unsigned char *State* = SetI2c1(unsigned char *Speed*,
 unsigned char *Pin*,
 unsigned short *Address*,
 unsigned char *GeneralCallEnable*,
 unsigned char *AddressMode*);

設定 I2C1 操作模式並啟動(請參考：[I/O 功能腳位對應表](#))。

- *Speed* 常數或變數值(0~2)，設定運行速度，0: 100kbps，1: 400kbps，2: 1MHzbps。預設值為 0
- *Pin* 常數或變數值(0~2)，設定 SCL 及 SDA 腳位，0: PC0 及 PC1，1: PC6 及 PC7，2: PE9 及 PE10。預設值為 0

<i>Mode</i>	SCL 及 SDA
0	PC0 及 PC1
1	PC6 及 PC7

- **Address** 常數或變數值(0~1023) · 設定在 Slave 模式時之 Slave ID · 7bit Address: 0~127 · 10bit Address: 0~1023。
- **GeneralCallEnable** 常數或變數值(0~1) · 設定 General Call 啟動或取消(Slave ID=0) · 0: Disable · 1: Enable · 預設值為 0
- **AddressMode** 常數或變數值(0~1) · 設定 7bit 或 10bit Address 模式 · 0: 7bit Address · 1: 10bit Address · 預設值為 0
- **State** 回傳變數值(0~255) · 0: 無狀況 · 255: 設定失敗。

```
unsigned char State = I2c1MasterTransmit( void *Array,
                                         unsigned short Length,
                                         unsigned short Address,
                                         unsigned char AddressMode);
```

I2C1 Master 模式資料輸出功能。

- **Array** 指標變數 · 傳遞所要輸出之變數記憶體位址。
- **Length** 常數或變數值(0~65535) · 傳遞所要輸出之變數長度 · 設定 0 或 1 皆為長度 1 · 預設值為 0
- **Address** 常數或變數值(0~1023) · 設定 Target I2C Slave ID · 7bit Address: 0~127 · 10bit Address: 0~1023。
- **AddressMode** 常數或變數值(0~1) · 設定 7bit 或 10bit Address 模式 · 0: 7bit Address · 1: 10bit Address · 預設值為 0
- **State** 回傳值(0~255) · 0: 無狀況 · 255: 設定失敗。

```
unsigned char State = I2c1MasterReceive(void *Array,
                                         unsigned short Length,
                                         unsigned short Address,
                                         unsigned char AddressMode);
```

I2C1 Master 模式資料輸入功能。

- **Array** 指標變數 · 傳遞所要輸入之變數記憶體位址。
- **Length** 常數或變數值(0~65535) · 傳遞所要輸入之變數長度 · 設定 0 或 1 皆為長度 1 · 預設值為 0
- **Address** 常數或變數值(0~1023) · 設定 Target I2C Slave ID · 7bit Address: 0~127 · 10bit Address: 0~1023。
- **AddressMode** 常數或變數值(0~1) · 設定 7bit 或 10bit Address 模式 · 0: 7bit Address · 1: 10bit Address · 預設值為 0
- **State** 回傳值(0~255) · 0: 無狀況 · 255: 設定失敗。

unsigned char **State** = GetI2c1MasterState(void);

讀取 I2C1 Master 模式狀態通知。

- **State** 回傳值(0~255) · 0: 無狀況 · 1: 資料傳輸完成 · 2: 資料傳輸失敗 · 3: 資料傳輸中 · 255: 讀取狀態錯誤。

unsigned char **State** = GetI2c1SlaveState(void);

讀取 I2C1 Slave 模式狀態通知。

- **State** 回傳值(0~255) · 0: 無狀況 · 1: 資料傳輸完成 · 2: 資料傳輸失敗(包括資料長度不符) · 3: 資料傳輸中 · 4: Slave 模式準備資料輸出 · 5: Slave 模式準備資料輸入 · 6: 準備 General Call 資料輸出 · 7: 準備 General Call 資料輸入 · 255: 讀取狀態錯誤。

unsigned char **State** = I2c1SlaveTransmit(void ***Array**, unsigned short **Length**);

I2C1 Slave 模式資料輸出功能。

- **Array** 指標變數 · 傳遞所要輸出之變數記憶體位址。
- **Length** 常數或變數值(0~65535) · 傳遞所要輸出之變數長度 · 設定 0 或 1 皆為長度 1 · 預設值為 0
- **State** 回傳值(0~255) · 0: 無狀況 · 255: 設定失敗。

unsigned char **State** = I2c1SlaveReceive(void ***Array**, unsigned short **Length**);

I2C1 Slave 模式資料輸入功能。

- **Array** 指標變數 · 傳遞所要入之變數記憶體位址
- **Length** 常數或變數值(0~65535) · 傳遞所要輸入之變數長度 · 設定 0 或 1 皆為長度 1 · 預設值為 0
- **State** 回傳值(0~255) · 0: 無狀況 · 255: 設定失敗。

unsigned char **State** = SetI2c1Off(void);

取消 I2C1 功能設定。

- **State** 回傳值(0~255) · 0: 取消成功 · 255: 取消失敗。

Event 相關設定

unsigned char **State** = SetI2c1Event(unsigned char **MasterEventEnable**,
unsigned char **SlaveEventEnable**);

設定 I2C 事件(Event)觸發功能。

- **MasterEventEnable** 常數或變數值(0~1) · 啟動或關閉 I2C1 Master 模式事件 · 0: Disable · 1: Enable。
- **SlaveEventEnable** 常數或變數值(0~1) · 啟動或關閉 I2C1 Slave 模式事件 · 0:

Disable · 1: Enable ·

- **State** 回傳值(0~255) · 0: 無狀況 · 255: 設定失敗。

void I2c1MasterEvent(void);

當 GetI2c1MasterState()回傳值轉變為 2(資料傳輸完成)或 3(資料傳輸失敗)時，系統所設定之程式執行區塊。

<i>state</i>	狀態說明
1	資料傳輸完成
2	資料傳輸失敗

void I2c1SlaveEvent(void);

當 GetI2c1SlaveState()回傳值轉變為 2(資料傳輸完成)、3(資料傳輸失敗)、4(Slave 模式準備資料輸出)、5(Slave 模式準備資料輸入)、6(準備 General Call 資料輸出)或 7(準備 General Call 資料輸入)時，系統所設定之程式執行區塊。

<i>state</i>	狀態說明
1	資料傳輸完成
2	資料傳輸失敗
4	Slave 模式準備資料輸出
5	Slave 模式準備資料輸入
6	準備 General Call 資料輸出
7	準備 General Call 資料輸入

```
#include "arminno.h"
#define EEAddr 0x51
unsigned char DataBuf[12];
int main(void) {
    SetI2C1(0, 1, 0, 0);
    // Write EEPROM
    DataBuf[0] = 0;
    DataBuf[1] = 0xA0;
    DataBuf[2] = 0x23;
    DataBuf[3] = 0x11;
    I2C1MasterTransmit(DataBuf, 4, EEAddr, 0);
    while(GetI2C1MasterState() == 3);

    Pause(100);
    // Read EEPROM
```

```

    DataBuf[0] = 0;
    DataBuf[1] = 0;
    I2C1MasterTransmit(DataBuf, 1, EEAddr, 0);
    while(GetI2C1MasterState() == 3);

    Pause(50);

    DataBuf[0] = DataBuf[1] = 0xAA;
    while(I2C1MasterReceive(DataBuf, 3, EEAddr, 0));
    while(GetI2C1MasterState() == 3);

    if((DataBuf[0] != 0xA0) || (DataBuf[1] != 0x23) ||
        (DataBuf[2] != 0x11))
        printf("Error\r\n");
    while(1);
}

```

SPI0

序列周邊介面 SPI0(Serial Peripheral Interface Bus) · 使用主從架構(Master/Slave) · 其使用 4 條實體資料傳輸線(SEL、SCK、MISO 及 MOSI) · 可設定成 open drain 加上上拉電阻或 push pull 模式交換串列資訊 · 透過 SEL 操作可進行一對一及一對多模組資料傳輸。

Arminno™上之 SPI0 函式庫功能特性如下：

- 允許 SPI0 工作於 3.3V 及 5V
- 主從模式切換(Master or Slave)
- Master 模式最快 36MHz 之傳輸速度設定
- Slave 模式最快 24MHz 之傳輸速度設定
- 可切換三組不同輸出入 I/O (請參考：[I/O 功能腳位對應表](#))

函式庫功能介紹如下：

名稱	說明
SetSpi0()	設定 SPI0 相關參數並啟動
Spi0MasterStart()	SPI0 Master 模式下啟動 SEL Active
Spi0MasterTransmit()	SPI0 Master 資料輸出
Spi0MasterReceive()	SPI0 Master 資料輸入
Spi0MasterStart()	SPI0 Master 模式下啟動 SEL Active
GetSpi0MasterState()	讀取 SPI0 Master 狀態
GetSpi0SlaveState()	讀取 SPI0 Slave 狀態
Spi0SlaveTransmit()	SPI0 Slave 資料輸出
Spi0SlaveReceive()	SPI0 Slave 資料輸入
SetSpi0Off()	取消 SPI0 設定
SPI 事件 (Event) 設定	
SetSpi0Event()	設定 SPI0 Event 功能
Spi0MasterEvent()	SPI0 Master 資料觸發事件
Spi0SlaveEvent()	SPI0 Slave 資料觸發事件

指令詳細說明

```
unsigned char State = SetSpi0(unsigned short ClkPrescaler,  
                                unsigned char OutputMode,  
                                unsigned char ControlMode,  
                                unsigned char CommunicationMode,  
                                unsigned char BitMode);
```

設定 SPI0 操作模式並啟動。

- **ClkPrescaler** 常數或變數值(0~65535) · SPI0 運行時脈除頻參數，可設定 SPI0 運行速度，公式為： $\text{SPI0 運行時脈速度} = 72\text{MHz} / (2 \times (\text{ClkPrescaler} + 1))$ 。預設值為 359，SPI0 運行在 100KHz。
- **OutputMode** 常數或變數值(0~5) · SPI0 通訊腳位模式設定，模式選擇如下：預設值為 0。

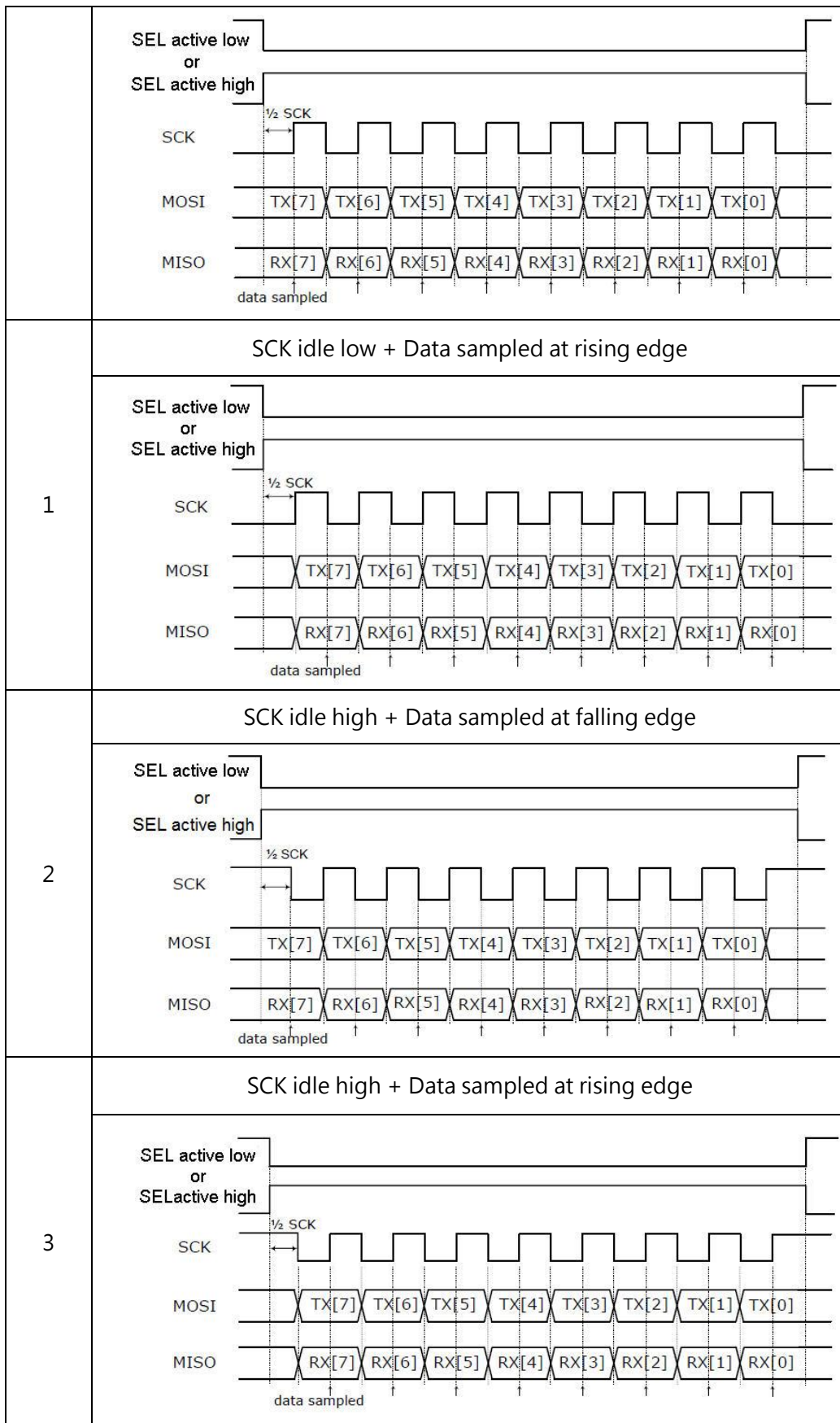
模式	通訊腳位				I/O 模式	工作電壓
	SEL	SCK	MOSI	MISO		
0	PB0	PB1	PB2	PB3	Push Pull	3.3V、5V
1	PB0	PB1	PB2	PB3	Open drain	3.3V、5V
2	PD0	PD1	PD2	PD3	Push pull	3.3V、5V
3	PD0	PD1	PD2	PD3	Open drain	3.3V、5V
4	PD8	PD9	PD10	PD11	Push pull	3.3V、5V
5	PD8	PD9	PD10	PD11	Open drain	3.3V、5V

- **ControlMode** 常數或變數值(0~3) · SPI 主從操作模式設定，模式選擇如下：預設值為 0。

模式	主從關係設定	SEL 腳位 Active level 設定
0	Master	LOW
1	Master	HIGH
2	Slave	LOW
3	Slave	HIGH

- **CommunicationMode** 常數或變數值(0~3) · SPI 通訊操作模式設定，模式選擇如下：預設值為 0。

模式	SEL 腳位 Active level 設定
0	SCK idle low + Data sampled at falling edge



- **BitMode** 常數或變數值(0~31)·設定所要傳輸資料的位元順序及每筆資料位元數。預設值為 23。

BitMode	傳送位元 順序	每筆資料 位元數	BitMode	傳送位元 順序	每筆資料 位元數
0	LSB	1	16	MSB	1
1	LSB	2	17	MSB	2
2	LSB	3	18	MSB	3
3	LSB	4	19	MSB	4
4	LSB	5	20	MSB	5
5	LSB	6	21	MSB	6
6	LSB	7	22	MSB	7
7	LSB	8	23	MSB	8
8	LSB	9	24	MSB	9
9	LSB	10	25	MSB	10
10	LSB	11	26	MSB	11
11	LSB	12	27	MSB	12
12	LSB	13	28	MSB	13
13	LSB	14	29	MSB	14
14	LSB	15	30	MSB	15
15	LSB	16	31	MSB	16

- **State** 回傳值(0~255)·0:無狀況·255:設定失敗。

unsigned char **State** = Spi0MasterStart(void);

SPI0 Master 模式下啟動 SEL Active。

- **State** 回傳值(0~255)·0:啟動成功·255:啟動失敗。

unsigned char **State** = Spi0MasterTransmit(void ***Array**, unsigned short **Length**);

SPI0 Master 模式資料輸出功能。

- **Array** 指標變數·傳遞所要輸出之變數記憶體位址。
- **Length** 常數或變數值(0~65535)·傳遞所要輸出之變數長度。
- **State** 回傳值(0~255)·0:無狀況·255:設定失敗。

unsigned char **State** = Spi0MasterReceive(void ***RxArray**,
unsigned short **RxLength**);

SPI0 Master 模式資料讀取功能。

- **RxArray** 指標變數，傳遞所要讀取之變數記憶體位址。
- **RxLength** 常數或變數值(0~65535)，傳遞所要讀取之變數長度。
- **State** 回傳值(0~255)，0:無狀況，255:設定失敗。

unsigned char **State** = Spi0MasterStop(void);

SPI0 Master 模式下取消 SEL Inactive。

- **State** 回傳值(0~255)，0:取消成功，255:取消失敗。

unsigned char **State** = GetSpi0MasterState(void);

讀取 SPI0 Master 模式之狀態情況。

- **State** 回傳值(0~255)，0: 無狀況，1: 資料傳輸完成，2:資料傳輸失敗，3:資料傳輸中，255: 讀取狀態錯誤。

```
#include "arminno.h"
unsigned char Buffer[32];
// PB0, PB1, PB2, PB3
int main(void) {
    SetSpi0(359, 0, 0, 0, 23);
    while(1) {
        Spi0MasterStart();
        Spi0MasterReceive(Buffer, 2);
        while(GetSpi0MasterState() == 3);
        Spi0MasterStop();
        float fValue = (Buffer[0]*256 + Buffer[1])/8*0.0625;
        printf("%.4fdeg C\r\n", fValue);
        Pause(10000);
    }
}
```

unsigned char **State** = Spi0SlaveReceive(void ***Array**, unsigned short **Length**);

SPI0 Slave 模式資料輸入功能。SPI0 內建 8 Level Frame 緩衝暫存，使用者可於 Data start 發生之後在讀取 8 Level Frame 緩衝時間內執行此函式，皆可正確接收資料。此函式可在傳輸發生前預先被執行等待資料傳輸。

- **Array** 指標變數，傳遞所要輸入之變數記憶體位址。
- **Length** 常數或變數值(0~65535)，傳遞所要輸入之變數長度。
- **State** 回傳值(0~255)，0:無狀況，255:設定失敗。

unsigned char **State** = Spi0SlaveTransmit(void ***Array**, unsigned short **Length**);

SPI0 Slave 模式資料輸出功能。在 Data start 發生之後執行此函式，則資料傳送會依據 SPI0 運作速度快慢，從第 2 個 frame 或之後的 frame 開始傳送資料。若是要在一開始傳輸就回傳資料，則必須在 Data start 發生之前就執行此函式功能。此函式可在傳輸發生前預先被執行等待資料傳輸。

- **Array** 指標變數，傳遞所要輸出之變數記憶體位址
- **Length** 常數或變數值(0~65535)，傳遞所要輸出之變數長度
- **State** 回傳值(0~255)，0:傳輸成功，255:設定失敗。

unsigned char **State** = GetSpi0SlaveState(unsigned char **RxFifoLevel**);

讀取 SPI0 Slave 模式之狀態情況。

- **RxFifoLevel** 回傳值(0~255)，RxFIFO 資料暫存筆數，0~8:資料暫存筆數，9: 資料溢位。
- **State** 回傳值(0~255)，0:無狀況，1:資料傳輸完成，2:資料傳輸失敗，3:資料傳輸中，4:Data start，255:讀取失敗。

unsigned char **State** = ResetSpi0SlaveRxFifo(void);

清除 SPI0 Slave 模式之 RxFIFO 暫存資料。

- **State** 回傳值(0~255)，0:設定成功，255:設定失敗。

unsigned char **State** = SetSpi0Off(void);

取消 SPI0 模式設定。

- **State** 回傳值(0~255)，0: 取消成功，255: 取消失敗。

Event 相關設定

unsigned char **State** = SetSpi0Event(unsigned char **MasterEvent**,
unsigned char **SlaveEvent**);

設定 SPI0 事件(Event)觸發功能。

- **MasterEvent** 常數或變數值(0~1)，啟動或關閉 SPI0 Master 模式觸發事件(Event)，0: Disable，1: Enable。
- **SlaveEvent** 常數或變數值(0~1)，啟動或關閉 SPI0 Slave 模式觸發事件(Event)，0: Disable，1: Enable。
- **State** 回傳值(0~255)，0: 無狀況，255: 設定失敗。

void Spi0MasterEvent(void);

當 GetSpi0MasterState()回傳值轉變為 1(資料傳輸完成)或 2(資料傳輸失敗)時，系統所設定之程式執行區塊。

<i>state</i>	狀態說明
--------------	------

1	資料傳輸完成
2	資料傳輸失敗

void Spi0SlaveEvent(void);

當 GetSpi0SlaveState()回傳值轉變為 1(資料傳輸完成)、2(資料傳輸失敗)及 4:DataStart 時系統所設定之程式執行區塊。

<i>state</i>	狀態說明
1	資料傳輸完成
2	資料傳輸失敗
4	Data start

SPI1

序列周邊介面 SPI1(Serial Peripheral Interface Bus) · 使用主從架構(Master/Slave) · 其使用 4 條實體資料傳輸線(SEL、SCK、MISO 及 MOSI) · 可設定成 open drain 加上上拉電阻或 push pull 模式交換串列資訊 · 透過 SEL 操作可進行一對一及一對多模組資料傳輸。

Arminno™上之 SPI1 函式庫功能特性如下：

- 允許 SPI1 工作於 3.3V 及 5V
- 主從模式切換(Master or Slave)
- Master 模式最快 36MHz 之傳輸速度設定
- Slave 模式最快 24MHz 之傳輸速度設定
- 可切換五組不同輸出入 I/O (請參考：[I/O 功能腳位對應表](#))

函式庫功能介紹如下：

名稱	說明
SetSpi1()	設定 SPI1 相關參數並啟動
Spi1MasterStart()	SPI1 Master 模式下啟動 SEL Active
Spi1MasterTransmit()	SPI1 Master 資料輸出
Spi1MasterReceive()	SPI1 Master 資料輸入
Spi1MasterStop()	SPI1 Master 模式下取消 SEL Active
GetSpi1MasterState()	讀取 SPI1 Master 狀態
GetSpi1SlaveState()	讀取 SPI1 Slave 狀態
Spi1SlaveTransmit()	SPI1 Slave 資料輸出
Spi1SlaveReceive()	SPI1 Slave 資料輸入
SetSpi1Off()	取消 SPI1 設定
SPI 事件 (Event) 設定	
SetSpi1Event()	設定 SPI1 Event 功能
Spi1MasterEvent()	SPI1 Master 資料觸發事件
Spi1SlaveEvent()	SPI1 Slave 資料觸發事件

指令詳細說明

```
unsigned char State = SetSpi1(unsigned short ClkPrescaler,  
                                unsigned char OutputMode,  
                                unsigned char ControlMode,  
                                unsigned char CommunicationMode,  
                                unsigned char BitMode);
```

設定 SPI1 操作模式並啟動。

- **ClkPrescaler** 常數或變數值(0~65535) · SPI1 運行時脈除頻參數，可設定 SPI1 運行速度，公式為： $\text{SPI1 運行時脈速度} = 72\text{MHz} / (2 \times (\text{ClkPrescaler} + 1))$ 。預設值為 359，SPI1 運行在 100KHz。
- **OutputMode** 常數或變數值(0~9) · SPI1 通訊腳位模式設定，模式選擇如下：
預設值為 0。

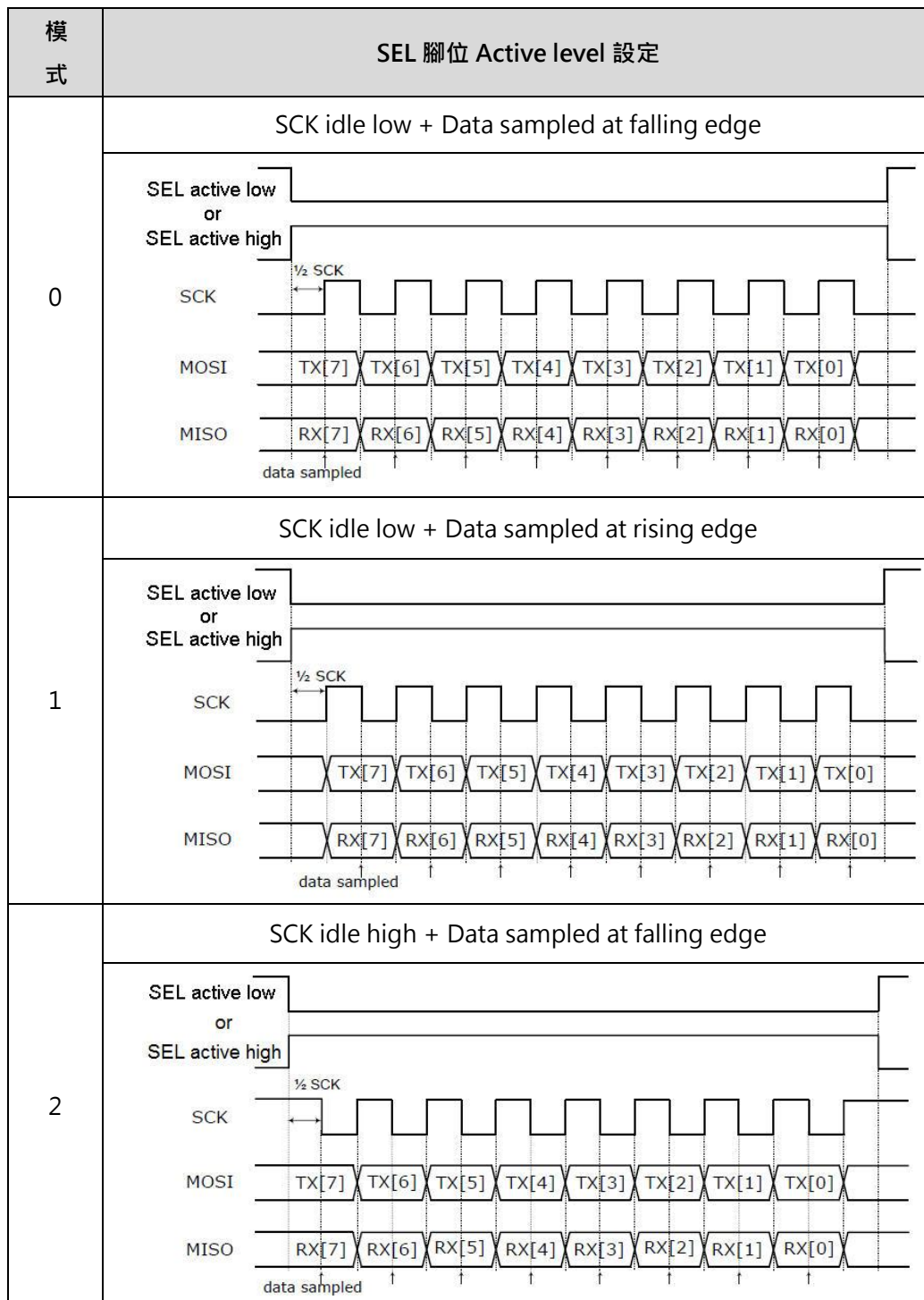
模 式	通訊腳位				I/O 模式	工作電 壓
	SEL	SCK	MOSI	MISO		
0	PA4	PA5	PA6	PA7	Push Pull	3.3V
1	PA4	PA5	PA6	PA7	Open drain	3.3V
2	PA8	PA9	PA10	PA11	Push pull	3.3V 5V
3	PA8	PA9	PA10	PA11	Open drain	3.3V 5V
4	PC0	PC1	PC2	PC3	Push pull	3.3V 5V
5	PC0	PC1	PC2	PC3	Open drain	3.3V 5V
6	PD4	PD5	PD6	PD7	Push Pull	3.3V 5V
7	PD4	PD5	PD6	PD7	Open drain	3.3V 5V
8	PE5	PE6	PE7	PE8	Push pull	3.3V
9	PE5	PE6	PE7	PE8	Open drain	3.3V

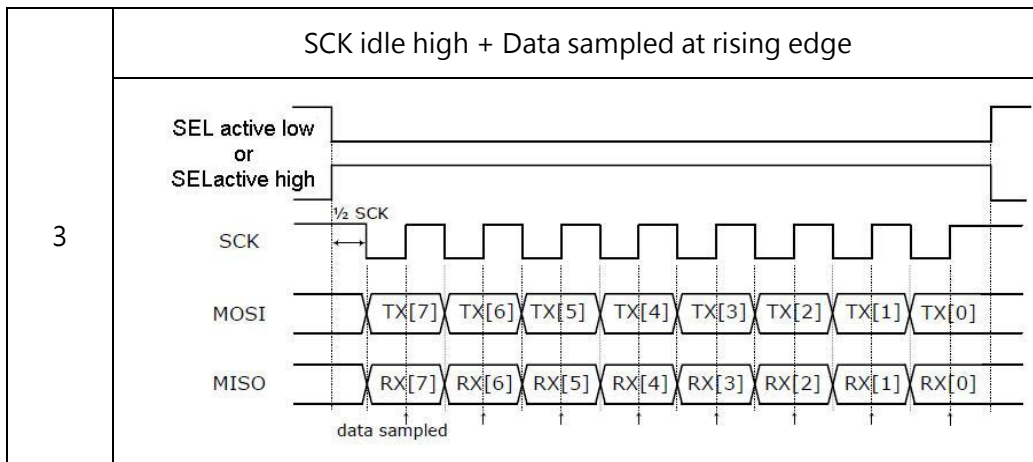
- **ControlMode** 常數或變數值(0~3) · SPI 主從操作模式設定，模式選擇如下：
預設值為 0。

模 式	主從關係 設定	SEL 腳位 Active level 設定
0	Master	LOW

1	Master	HIGH
2	Slave	LOW
3	Slave	HIGH

- **CommunicationMode** 常數或變數值(0~3) · SPI 通訊操作模式設定 · 模式選擇如下：預設值為 0。





- **BitMode** 常數或變數值(0~31) , 設定所要傳輸資料的位元順序及每筆資料位元數。預設值為 23。

<i>BitMode</i>	傳送位元 順序	每筆資料 位元數	<i>BitMode</i>	傳送位元 順序	每筆資料 位元數
0	LSB	1	16	MSB	1
1	LSB	2	17	MSB	2
2	LSB	3	18	MSB	3
3	LSB	4	19	MSB	4
4	LSB	5	20	MSB	5
5	LSB	6	21	MSB	6
6	LSB	7	22	MSB	7
7	LSB	8	23	MSB	8
8	LSB	9	24	MSB	9
9	LSB	10	25	MSB	10
10	LSB	11	26	MSB	11
11	LSB	12	27	MSB	12
12	LSB	13	28	MSB	13
13	LSB	14	29	MSB	14
14	LSB	15	30	MSB	15
15	LSB	16	31	MSB	16

- **State** 回傳值(0~255) , 0:無狀況 , 255:設定失敗。

unsigned char **State** = Spi1MasterStart(void);

SPI1 Master 模式下啟動 SEL Active。

- **State** 回傳值(0~255) · 0:啟動成功 · 255:啟動失敗。

unsigned char **State** = Spi1MasterTransmit(void ***Array**, unsigned short **Length**);

SPI1 Master 模式資料輸出功能。

- **Array** 指標變數 · 傳遞所要輸出之變數記憶體位址。
- **Length** 常數或變數值(0~65535) · 傳遞所要輸出之變數長度。
- **State** 回傳值(0~255) · 0:無狀況 · 255:設定失敗。

unsigned char **State** = Spi1MasterReceive(void ***RxArray**,
unsigned short **RxLength**);

SPI1 Master 模式資料讀取功能。

- **RxArray** 指標變數 · 傳遞所要讀取之變數記憶體位址。
- **RxLength** 常數或變數值(0~65535) · 傳遞所要讀取之變數長度。
- **State** 回傳值(0~255) · 0:無狀況 · 255:設定失敗。

unsigned char **State** = Spi1MasterStop(void);

SPI1 Master 模式下取消 SEL Inactive。

- **State** 回傳值(0~255) · 0:取消成功 · 255:取消失敗。

unsigned char **State** = GetSpi1MasterState(void);

讀取 SPI1 Master 模式之狀態情況。

- **State** 回傳值(0~255) · 0: 無狀況 · 1: 資料傳輸完成 · 2:資料傳輸失敗 · 3: 資料傳輸中 · 255: 讀取狀態錯誤。

```
#include "arminno.h"
unsigned char Buffer[32];
int main(void) {
    SetSpi1(359, 0, 0, 0, 23); // PA4, PA5, PA6, PA7
    while(1) {
        Spi1MasterStart();
        Spi1MasterReceive(Buffer, 2);
        while(GetSpi1MasterState() == 3);
        Spi1MasterStop();
        float fValue = (Buffer[0]*256 + Buffer[1])/8*0.0625;
        printf("%.4fdeg C\r\n", fValue);
        Pause(10000);
    }
}
```

unsigned char **State** = Spi1SlaveReceive(void ***Array**, unsigned short **Length**);

SPI1 Slave 模式資料輸入功能。SPI1 內建 8 Level Frame 緩衝暫存，使用者可於 Data start 發生之後在讀取 8 Level Frame 緩衝時間內執行此函式，皆可正確接收資料。此函式可在傳輸發生前預先被執行等待資料傳輸。

- **Array** 指標變數，傳遞所要輸入之變數記憶體位址。
- **Length** 常數或變數值(0~65535)，傳遞所要輸入之變數長度。
- **State** 回傳值(0~255)，0:無狀況，255:設定失敗。

unsigned char **State** = Spi1SlaveTransmit(void ***Array**, unsigned short **Length**);

SPI1 Slave 模式資料輸出功能。在 Data start 發生之後執行此函式，則資料傳送會依據 SPI1 運作速度快慢，從第 2 個 frame 或之後的 frame 開始傳送資料。若是要在一開始傳輸就回傳資料，則必須在 Data start 發生之前就執行此函式功能。此函式可在傳輸發生前預先被執行等待資料傳輸。

- **Array** 指標變數，傳遞所要輸出之變數記憶體位址
- **Length** 常數或變數值(0~65535)，傳遞所要輸出之變數長度
- **State** 回傳值(0~255)，0:傳輸成功，255:設定失敗。

unsigned char **State** = GetSpi1SlaveState(unsigned char **RxFifoLevel**);

讀取 SPI1 Slave 模式之狀態情況。

- **RxFifoLevel** 回傳值(0~255)，RxFIFO 資料暫存筆數，0~8:資料暫存筆數，9: 資料溢位。
- **State** 回傳值(0~255)，0:無狀況，1:資料傳輸完成，2:資料傳輸失敗，3:資料傳輸中，4:Data start，255:讀取失敗。

unsigned char **State** = ResetSpi1SlaveRxFifo(void);

清除 SPI1 Slave 模式之 RxFIFO 暫存資料。

- **State** 回傳值(0~255)，0:設定成功，255:設定失敗。

unsigned char **State** = SetSpi1Off(void);

取消 SPI1 模式設定。

- **State** 回傳值(0~255)，0: 取消成功，255: 取消失敗。

Event 相關設定

unsigned char **State** = SetSpi1Event(unsigned char **MasterEvent**,
unsigned char **SlaveEvent**);

設定 SPI1 事件(Event)觸發功能。

- **MasterEvent** 常數或變數值(0~1)，啟動或關閉 SPI1 Master 模式觸發事件

(Event) · 0: Disable · 1: Enable。

- **SlaveEvent** 常數或變數值(0~1) · 啟動或關閉 SPI1 Slave 模式觸發事件(Event) · 0: Disable · 1: Enable。
- **State** 回傳值(0~255) · 0: 無狀況 · 255: 設定失敗。

void Spi1MasterEvent(void);

當 GetSpi1MasterState()回傳值轉變為 1(資料傳輸完成)或 2(資料傳輸失敗)時，系統所設定之程式執行區塊。

<i>state</i>	狀態說明
1	資料傳輸完成
2	資料傳輸失敗

void Spi1SlaveEvent(void);

當 GetSpi1SlaveState()回傳值轉變為 1(資料傳輸完成)、2(資料傳輸失敗)及 4:Data Start 時系統所設定之程式執行區塊。

<i>state</i>	狀態說明
1	資料傳輸完成
2	資料傳輸失敗
4	Data start

UART0

通用非同步接收/傳送器 UART0(Universal Asynchronous Receiver/Transmitter) 為一種非同步串列通信口的總稱，它包括了 RS232、RS422、RS423、RS449 和 RS485 等介面標準規範和匯流排標準規範。規定了通信口的電力特性、傳輸速率、連接特性和介面的機械特性等內容。UART0 為當今最普遍的串列傳輸方式之一，幾乎所有軟硬體都可支援 UART 通訊協定。Arminno™內建 UART 函式庫功能特性如下：

- 允許 UART0 工作於 3.3V 及 5V
- 可切換 4 組不同輸出 I/O (請參考：I/O 功能腳位對應表)
- 300bps~1Mbps 傳輸速度設定
- 16Level TxFIFO 與 16Level RxFIFO 設定
- MSB 或 LSB 開始傳送功能設定
- 1 或 2 個停止位元(stop bit)設定
- 偶數、奇數和無同位元(parity)設定
- RxFIFO 累積資料觸發事件

函式庫功能如下：

名稱	說明
SetUart0()	設定 UART0 相關參數並啟動
GetUart0TxState()	讀取 UART0 TX 資料輸出狀態
GetUart0RxState()	讀取 UART0 RX 資料輸入狀態
SendUart0Data()	將資料由 UART0 TX 輸出
GetUart0Data()	從 UART0 RX 讀取資料
ResetUart0RxFifo()	清除 UART0 RxFIFO 暫存資料
SetUart0Off()	取消 UART0 設定
UART 事件設定	
SetUart0Event()	設定 UART0 事件功能
Uart0TxEvent()	UART0 TX 資料傳送完畢事件
Uart0RxEvent()	UART0 RX 輸入資料觸發事件

指令詳細說明：

```
unsigned char State = SetUart0(unsigned long Baudrate,  
                                unsigned char PIN,  
                                unsigned char OutputMode,  
                                unsigned char Parity,
```

unsigned char *StopBit*,
 unsigned char *RxBufferTriggerLevel*,
 unsigned char *BitMode*);

設定 UART0 相關參數及啟動。

- **Baudrate** 常數或變數值(300~1,000,000) · 設定 UART0 Baudrate 參數 · 單位 bps
- **PIN** 常數或變數值(0~3) · UART0 通訊腳位模式設定 · 模式選擇如下：

模 式	TX	RX	工作電壓
0	PA2	PA3	3.3V
1	PA10	PA11	3.3V/5V
2	PB10	PB13	3.3V/5V
3	PC8	PC10	3.3V/5V

- **OutputMode** 常數或變數值(0~3) · 選擇通訊傳輸模式 · 模式選擇如下：預設值為 0

模式	輸出設定	工作電壓
0	Push pull	-
1	Open drain	-
2	RS485 模式(啟動 RTS 腳位 PA8)	3.3V/5V
3	RS485 模式(啟動 RTS 腳位 PB8)	3.3V/5V
4	RS485 模式(啟動 RTS 腳位 PC11)	3.3V/5V

- **Parity** 常數或變數值(0~2) · 設定位元檢查模式 · 0: No Parity · 1: Parity Even · 2: Parity Odd · 預設值為 0。
- **StopBit** 常數或變數值(0~2) · 設定位元檢查模式 · 0: 1 Stop Bit · 1: 2 Stop Bit · 預設值為 0。
- **RxBufferTriggerLevel** 常數或變數值(0~3) · 設定已接收 RxFIFO 資料長度觸發位準 · 0: Disable · 1: 1byte · 2: 4byte · 3: 8byte · 4: 14byte · 預設值為 0。
- **BitMode** 常數或變數值(0~1) · 設定位元檢查模式 · 0: LSB first · 1: MSB first · 預設值為 0。
- **State** 回傳值(0~255) · 0: 設定成功 · 255: 設定失敗。

unsigned char **State** = SendUart0Data(void **Array*, unsigned short *Length*);
 UART0 TX 資料輸出。

- **Array** 指標變數，傳遞所要輸出之變數記憶體位址。
- **Length** 常數或變數值(0~65535)，傳遞所要輸出之變數長度。
- **State** 回傳值(0~255)，0: 無狀況，255:設定失敗。

unsigned char **State** = **GetUart0TxState**(void);

讀取 UART0 TX 狀態通知。

- **State** 回傳值(0~255)，0: 無狀況，1:資料輸出完成，2:資料傳輸失敗，3: 資料正在輸出中，255: 讀取失敗。

```
#include "arminno.h"
unsigned char TxData = 0;
int main(void)
{
    SetUart0(38400, 0);    // PA2, PA3
    while(1) {
        SendUart0Data(&TxData, 1);
        TxData++;
        while(GetUart0TxState() == 3);
        Pause(2000);
    }
}
```

unsigned short **State** = **GetUart0Data**(void ***Array**, unsigned short **Length**);

讀取 UART0 RX 資料。UART0 內建 16 Level RxFIFO 緩衝暫存，使用者可於資料傳輸開始後之累積暫存 16 Level RxFIFO 緩衝時間內執行此函式。此函式可在傳輸發生前預先被執行等待資料傳輸。

- **Array** 指標變數，傳遞所要輸入之變數記憶體位址。
- **Length** 常數或變數值(0~65535)，傳遞所要輸入之變數長度。
- **State** 回傳值(0~255)，0: 無狀況，255:設定失敗。

unsigned char **State** = **GetUart0RxState**(unsigned char **RxFifoLevel**);

讀取 UART0 Rx 狀態通知。

- **RxFifoLevel** 回傳值(0~255)，RxFIFO 資料暫存筆數，0~16:資料暫存筆數，17: 資料溢位。
- **State** 回傳值(0~255)，0: 無狀況，1:資料傳輸完成，2:資料傳輸失敗，3:資料傳輸中，4: RxFIFO 資料長度位準觸發，255: 讀取失敗。

unsigned char **State** = ResetUart0RxFifo(void);

清除 UART0 Slave 模式之 RxFIFO 暫存資料。

- **State** 回傳值(0~255) · 0:設定成功 · 255:設定失敗。

unsigned char **State** = SetUart0Off(void);

取消 UART0 設定。

- **State** 回傳值(0~255) · 0: 設定成功 · 255: 設定失敗。

Event 相關設定

unsigned char **State** = SetUart0Event(unsigned char **Uart0TxEvent**,
unsigned char **Uart0RxEvent**);

設定 UART0 事件(Event)觸發功能。

- **Uart0TxEvent** 常數或變數值(0~1) · 啟動或關閉 Uart0TxEvent · 0: Disable · 1: Enable。
- **Uart0RxEvent** 常數或變數值(0~1) · 啟動或關閉 Uart0RxEvent · 0: Disable · 1: Enable。
- **State** 回傳值(0~255) · 0: 無狀況 · 255: 設定失敗。

void Uart0TxEvent (void);

當 GetUart0TxState()回傳值轉變為 1(資料傳輸完成)或 2(資料傳輸失敗)時，系統所設定之程式執行區塊。

<i>state</i>	狀態說明
1	資料傳輸完成
2	資料傳輸失敗

void Uart0RxEvent (void);

當 GetUart0RxState()回傳值轉變為 1(資料傳輸完成)、2(資料傳輸失敗)及 4: RxFIFO 資料長度位準觸發時系統所設定之程式執行區塊。

<i>state</i>	狀態說明
1	資料傳輸完成
2	資料傳輸失敗
4	RxFIFO 資料長度位準觸發

```
#include "arminno.h"
unsigned char TxData = 0;
volatile unsigned char TxDone ;
void Uart0TxEvent (void)
```

```

{
    if(GetUart0TxState() == 1) {
        TxDone = 1;
    }
}
int main(void)
{
    SetUart0(38400, 0);    // PA2, PA3
    SetUart0Event(1, 0);
    while(1) {
        TxDone = 0;
        SendUart0Data(&TxData, 1);
        while(TxDone == 0);
        TxData++;
        Pause(2000);
    }
}

```

UART1

通用非同步接收/傳送器 UART1(Universal Asynchronous Receiver/Transmitter) 為一種非同步串列通信口的總稱，它包括了 RS232、RS422、RS423、RS449 和 RS485 等介面標準規範和匯流排標準規範。規定了通信口的電力特性、傳輸速率、連接特性和介面的機械特性等內容。UART 為當今最普遍的串列傳輸方式之一，幾乎所有軟硬體都可支援 UART 通訊協定。Arminno™內建 UART 函式庫功能特性如下：

- 允許 UART1 工作於 3.3V 及 5V
- 可切換 4 組不同輸出 I/O (請參考：I/O 功能腳位對應表)
- 300bps~1Mbps 傳輸速度設定
- 16Level TxFIFO 與 16Level RxFIFO 設定
- MSB 或 LSB 開始傳送功能設定
- 1 或 2 個停止位元(stop bit)設定
- 偶數、奇數和無同位元(parity)設定
- RxFIFO 累積資料觸發事件

函式庫功能如下：

名稱	說明
SetUart1()	設定 UART1 相關參數並啟動
GetUart1TxState()	讀取 UART1 TX 資料輸出狀態
GetUart1RxState()	讀取 UART1 RX 資料輸入狀態
SendUart1Data()	將資料由 UART1 TX 輸出
GetUart1Data()	從 UART1 RX 讀取資料
ResetUart1RxFifo()	清除 UART1 RxFIFO 暫存資料
SetUart1Off()	取消 UART1 設定
UART 事件設定	
SetUart1Event()	設定 UART1 事件功能
Uart1TxEvent()	UART1 TX 資料傳送完畢事件
Uart1RxEvent()	UART1 RX 輸入資料觸發事件

指令詳細說明：

```
unsigned char State = SetUart1(unsigned long Baudrate,  
                                unsigned char PIN,  
                                unsigned char OutputMode,  
                                unsigned char Parity,
```

```

unsigned char StopBit,
unsigned char RxBufferTriggerLevel,
unsigned char BitMode);

```

設定 UART1 相關參數及啟動。

- **Baudrate** 常數或變數值(300~1,000,000) · 設定 UART1 Baudrate 參數 · 單位 bps
- **PIN** 常數或變數值(0~3) · UART1 通訊腳位模式設定 · 模式選擇如下：

模 式	TX	RX	工作電壓
0	PA6	PA7	3.3V
1	PB2	PB3	3.3V/5V
2	PC4	PC5	3.3V/5V
3	PE14	PE15	3.3V/5V

- **OutputMode** 常數或變數值(0~3) · 選擇通訊傳輸模式 · 模式選擇如下：預設值為 0

模式	輸出設定	工作電壓
0	Push pull	-
1	Open drain	-
2	RS485 模式(啟動 RTS 腳位 PA4)	3.3V
3	RS485 模式(啟動 RTS 腳位 PB0)	3.3V/5V
4	RS485 模式(啟動 RTS 腳位 PB15)	3.3V/5V
5	RS485 模式(啟動 RTS 腳位 PC13)	3.3V/5V

- **Parity** 常數或變數值(0~2) · 設定位元檢查模式 · 0: No Parity · 1: Parity Even · 2: Parity Odd · 預設值為 0。
- **StopBit** 常數或變數值(0~2) · 設定位元檢查模式 · 0: 1 Stop Bit · 1: 2 Stop Bit · 預設值為 0。
- **RxBufferTriggerLevel** 常數或變數值(0~3) · 設定已接收 RxFIFO 資料長度觸發位準 · 0: Disable · 1: 1byte · 2: 4byte · 3: 8byte · 4: 14byte · 預設值為 0。
- **BitMode** 常數或變數值(0~1) · 設定位元檢查模式 · 0: LSB first · 1: MSB first · 預設值為 0。
- **State** 回傳值(0~255) · 0: 設定成功 · 255: 設定失敗。

unsigned char **State** = SendUart1Data(void ***Array**, unsigned short **Length**);

UART1 TX 資料輸出。

- **Array** 指標變數，傳遞所要輸出之變數記憶體位址。
- **Length** 常數或變數值(0~65535)，傳遞所要輸出之變數長度。
- **State** 回傳值(0~255)，0: 無狀況，255:設定失敗。

unsigned char **State** = GetUart1TxState(void);

讀取 UART1 TX 狀態通知。

- **State** 回傳值(0~255)，0: 無狀況，1:資料輸出完成，2:資料傳輸失敗，3: 資料輸出中，255: 讀取失敗。

unsigned short **State** = GetUart1Data(void ***Array**, unsigned short **Length**);

讀取 UART1 RX 資料。UART1 內建 16 Level RxFIFO 緩衝暫存，使用者可於資料傳輸開始後之累積暫存 16 Level RxFIFO 緩衝時間內執行此函式。此函式可在傳輸發生前預先被執行等待資料傳輸。

- **Array** 指標變數，傳遞所要輸入之變數記憶體位址。
- **Length** 常數或變數值(0~65535)，傳遞所要輸入之變數長度。
- **State** 回傳值(0~255)，0: 無狀況，255:設定失敗。

unsigned char **State** = GetUart1RxState(unsigned char **RxFifoLevel**);

讀取 UART1 Rx 狀態通知。

- **RxFifoLevel** 回傳值(0~255)，RxFIFO 資料暫存筆數，0~16:資料暫存筆數，17: 資料溢位。
- **State** 回傳值(0~255)，0: 無狀況，1:資料傳輸完成，2:資料傳輸失敗，3:資料傳輸中，4: RxFIFO 資料長度位準觸發，255: 讀取失敗。

unsigned char **State** = ResetUart1RxFifo(void);

清除 UART1 Slave 模式之 RxFIFO 暫存資料。

- **State** 回傳值(0~255)，0:設定成功，255:設定失敗。

```
#include "arminno.h"
unsigned char RxFifoLevel ;
char Buffer[32];
int main(void)
{
    SetUart1(38400, 0);    // PA6, PA7
    ResetUart1RxFifo();
    while(1) {
```

```
GetUart1RxState(RxFifoLevel);
if(RxFifoLevel > 0) {
    GetUart1Data(Buffer, RxFifoLevel);
}
}
}
```

unsigned char *State* = SetUart1Off(void);

取消 UART1 設定。

- *State* 回傳值(0~255) · 0: 設定成功 · 255: 設定失敗。

Event 相關設定

unsigned char *State* = SetUart1Event(unsigned char *Uart1TxEvent*,
unsigned char *Uart1RxEvent*);

設定 UART1 事件(Event)觸發功能。

- *Uart1TxEvent* 常數或變數值(0~1) · 啟動或關閉 Uart1TxEvent · 0: Disable · 1: Enable。
- *Uart1RxEvent* 常數或變數值(0~1) · 啟動或關閉 Uart1RxEvent · 0: Disable · 1: Enable。
- *State* 回傳值(0~255) · 0: 無狀況 · 255: 設定失敗。

void Uart1TxEvent (void);

當 GetUart1TxState()回傳值轉變為 1(資料傳輸完成)或 2(資料傳輸失敗)時，系統所設定之程式執行區塊。

<i>state</i>	狀態說明
1	資料傳輸完成
2	資料傳輸失敗

void Uart1RxEvent (void);

當 GetUart1RxState()回傳值轉變為 1(資料傳輸完成)、2(資料傳輸失敗)及 4: RxFIFO 資料長度位準觸發時系統所設定之程式執行區塊。

<i>state</i>	狀態說明
1	資料傳輸完成
2	資料傳輸失敗
4	RxFIFO 資料長度位準觸發

USB

USB 設備控制器相容於 USB 2.0 Full Speed(12Mbps)規格。系統內建完全整合之 USB Full Speed 收發單元，擁有一個控制端點(EP0)用於控制傳輸，3 個單緩衝暫存(EP1~EP3)用於 bulk 及 interrupt 傳輸，4 個雙緩衝暫存(EP4~EP7)用於 bulk、interrupt 和 isochronous 傳輸，擁有 1,024bytes 端點資料緩衝區。USB 函式規劃模擬為 Virtual COM Port，易於規劃及使用。Arminno™內建 USB 函式庫功能特性如下：

- USB 2.0 Full Speed(12Mbps)規格
- 完全整合之 USB Full Speed 收發單元
- 1 個控制端點(EP0)用於控制傳輸
- 3 個單緩衝暫存(EP1~EP3)用於 bulk 及 interrupt 傳輸
- 4 個雙緩衝暫存(EP4~EP7)用於 bulk、interrupt 和 isochronous 傳輸
- 擁有 1,024bytes 端點資料緩衝區
- USB 函式規劃模擬為 Virtual COM Port(雙緩衝暫存,64byte buffer size)

函式庫功能如下：

名稱	說明
SetUsb()	設定 USB 相關參數並啟動
GetUsbTxState()	讀取 USB TX 資料輸出狀態
GetUsbRxState()	讀取 USB RX 資料輸入狀態
SendUsbData()	將資料由 USB TX 輸出
GetUsbData()	從 USB RX 讀取資料
SetUsbOff()	取消 USB 設定
USB 事件設定	
SetUsbEvent()	設定 USB 事件功能
UsbTxEvent()	USB TX 資料傳送完畢事件
UsbRxEvent()	USB RX 輸入資料觸發事件

指令詳細說明：

`unsigned char State = SetUsb(unsigned short RxBufferTriggerLevel);`

設定 USB 相關參數及啟動。

- *RxBufferTriggerLevel* 常數或變數值(0~512)，設定已接收 RxBuffer 資料長度觸發位準，0 和 1 接為 1byte 觸發。預設值為 0。
- *State* 回傳值(0~255)，0: 設定成功，255: 設定失敗。

unsigned char **State** = SendUsbData(void ***Array**, unsigned short **Length**);

USB TX 資料輸出。

- **Array** 指標變數，傳遞所要輸出之變數記憶體位址。
- **Length** 常數或變數值(0~65535)，傳遞所要輸出之變數長度。
- **State** 回傳值(0~255)，0: 無狀況，255:設定失敗。

unsigned char **State** = GetUsbTxState(void);

讀取 USB TX 狀態通知。

- **State** 回傳值(0~255)，0: 無狀況，1:資料輸出完成，2:資料傳輸失敗，3: 資料輸出中，255: 讀取失敗。

unsigned char **State** = GetUsbRxState(unsigned short **RxBufferLevel**);

讀取 USB Rx 狀態通知。

- **RxBufferLevel** 回傳值(0~512)，資料緩衝暫存筆數，0~512:資料暫存筆數，65535: 資料溢位。
- **State** 回傳值(0~255)·0: 無狀況·2:資料傳輸失敗·4:資料緩衝長度位準觸發，255: 讀取失敗。

unsigned char **State** = GetUsbData(void ***Array**, unsigned short **Length**);

讀取 USB RX 資料。內建 512byte 接收緩衝暫存，使用者可於資料傳輸開始後之累積暫存最大 512byte 資料緩衝時間內執行此函式開始資料接收，若資料筆數超過緩衝暫存大小則 GetUsbRxState()會回報資料傳輸失敗。

- **Array** 指標變數，傳遞所要輸入之變數記憶體位址。
- **Length** 常數或變數值(0~65535)，傳遞所要輸入之變數長度。
- **State** 回傳值(0~255)，0: 無狀況，255:設定失敗。

```
#include "arminno.h"
unsigned char DataBuf[] ={"Hello\r\n"};
unsigned short len;
int main(void)
{
    SetUsb(12);
    while(1) {
        SendUsbData(DataBuf, sizeof(DataBuf));
        while(GetUsbTxState() == 3);
        Pause(20000);
        GetUsbRxState(len);
        if(len) printf("%d\r\n", len);
    }
}
```

```

    }
}

```

unsigned char **State** = ResetUsbRxBuffer(void);

清除 USB 之 RxBuffer 暫存資料。

- **State** 回傳值(0~255) · 0:設定成功 · 255:設定失敗。

unsigned char **State** = SetUsbOff(void);

取消 USB 設定。

- **State** 回傳值(0~255) · 0: 設定成功 · 255: 設定失敗。

Event 相關設定

unsigned char **State** = SetUsbEvent(unsigned char *UsbTxEvent*,
unsigned char *UsbRxEvent*);

設定 USB 事件(Event)觸發功能。

- **UsbTxEvent** 常數或變數值(0~1) · 啟動或關閉 UsbTxEvent · 0: Disable · 1: Enable。
- **UsbRxEvent** 常數或變數值(0~1) · 啟動或關閉 UsbRxEvent · 0: Disable · 1: Enable。
- **State** 回傳值(0~255) · 0: 無狀況 · 255: 設定失敗。

void **UsbTxEvent** (void);

當 GetUsbTxState()回傳值轉變為 1(資料傳輸完成)或 2(資料傳輸失敗)時，系統所設定之程式執行區塊。

<i>state</i>	狀態說明
1	資料傳輸完成
2	資料傳輸失敗

void **UsbRxEvent** (void);

當 GetUsbRxState()回傳值轉變為 2(資料傳輸失敗)及 4: RxBuffer 資料長度位準觸發時系統所設定之程式執行區塊。

<i>state</i>	狀態說明
2	資料傳輸失敗
4	RxBuffer 資料長度位準觸發

Power Saving

電力節能指令對需要採用電池電力之移動手持式裝置可說是最重要的功能之一，如何在有限的電池電力中盡可能的延長移動手持式裝置持續使用時間，一直都是各廠商在開發手持式產品追逐的最重要目標。Arminno™提供節能相關常用函式功能如下：

名稱	說明
Sleep()	系統睡眠節能指令
GetResetState()	取得系統復位狀態

Arminno™提供了四種 Sleep Mode，其主要 IC HT32F1765 相關電力消耗數據提供如下表：

Mode	周邊狀態說明	電力消耗	喚醒方式
Run	Enable all	約 60 mA	---
	Disable all	約 27 mA	
Sleep	Enable all	約 42 mA	Any Event RTC wake-up Wake-up pin rising edge(PB6) 喚醒後從睡眠點繼續執行程式
	Disable all	約 9 mA	
Deep-Sleep1	Auto disable all (except RTC)	約 58 uA	GPIO Event RTC wake-up Wake-up pin rising edge(PB6) 喚醒後從睡眠點繼續執行程式
Deep-Sleep2	Auto disable all (except RTC)	約 18 uA	RTC wake-up Wake-up pin rising edge(PB6) 喚醒後從睡眠點繼續執行程式
Power Mode	Auto disable all (except RTC)	約 5u A	RTC wake-up Wake-up pin rising edge(PB6) External reset pin(nRST) 喚醒後從頭開始執行程式

指令詳細說明：

void Sleep(unsigned char Mode);

睡眠深度模式設定。

- **Mode** 常數或變數值(0~3)，節能模式設定。

Mode	狀態說明
------	------

0	Sleep
1	Deep sleep1
2	Deep sleep2
3	Power Down

unsigned char ***State*** = GetResetState(void);

讀取 Reset 後之狀態通知。

- ***State*** 回傳值(0~255) · 前次系統復位(reset)原因參數回傳。

<i>State</i>	狀態說明
0	Power on reset Software reset
1	Reset pin(nRST)
2	Power down mode
255	讀取失敗

```

#include "arminno.h"
// 注意! 一旦進入睡眠模式, e-link32 也無法燒錄或除錯
// 如果要清除或下載程式, 請按 reset 鍵, 並在未睡眠模式時(PA14
// LED 閃爍時) 清除或下載
int main(void)
{
    InitialGpioState(PA0, 0, 2);    // PA0 輸入 pull-high
    InitialGpioState(PA14, 1);
    SetGpioEvent(0, 0, 0, 0);      // PA0 設為 WAKEUP LOW
    while(1) {
        for(int i = 0 ; i < 5 ; i++) {
            Low(PA14);
            Pause(5000);
            High(PA14);
            Pause(5000);
        }
        Sleep(1); // 進入睡眠模式, 等待 PA0 = 0 喚醒
    }
}

```

Other commands

Arminno™提供其它相關常用函式庫功能如下：

名稱	說明
Pause()	時間延遲指令
SetMainClk()	切換主要 Clock 來源
SetX32Clk()	切換次要 32768Hz Clock 來源
GlobalEventControl()	開啟/關閉所有 Event 遮罩功能
GetLibVer()	讀取 Library 版本資訊
SetIdeOff()	取消 IDE 功能並釋放 I/O 腳位
GetSysTick()	讀取系統時鐘數值
SetClkOut()	設定時鐘源輸出功能

指令詳細說明：

void Pause(unsigned short *Time*);

設定延遲時間。

- ***Time*** 常數或變數值(0~65535)，延遲時間設定，單位: 100uS。

unsigned char *State* = SetMainClk(unsigned char *Mode*);

設定主要時鐘(8MHz)輸入來源，可切換內部時鐘源或外部時鐘源(佔用 PB11、PB12，需設定 Arminno™板上背面相關跳線短路)。

- ***Mode*** 常數或變數值(0~1)：0: 外部時鐘輸入(佔用 PB11 及 PB12 腳位)；1: 內部時鐘輸入，不佔用 GPIO 腳位。(請參考：[I/O 功能腳位對應表](#))。預設為 0
- ***State*** 回傳值(0~255)：0: 設定成功；255: 設定失敗。

unsigned char *State* = SetX32Clk(unsigned char *Mode*);

設定次要時鐘(32,768Hz)輸入來源，可切換內部時鐘源或外部時鐘源(佔用 PB4、PB5)。

- ***Mode*** 常數或變數值(0~1)：0: 外部時鐘輸入(佔用 PB4 及 PB5 腳位)；1: 內部時鐘輸入，不佔用 GPIO 腳位。(請參考：[I/O 功能腳位對應表](#))。預設為 0
- ***State*** 回傳值(0~255)：0: 設定成功；255: 設定失敗。

unsigned char *State* = SetIdeOff(void);

IDE 開發除錯功能在復位(reset)時預設為啟動狀態，取消 IDE 開發除錯功能，可釋放 PE11、PE12 及 PE13(GPIO 腳位)可供一般使用。(請參考：[I/O 功能腳位對應表](#))。

- ***State*** 回傳值(0~255)：0: 取消成功；255: 取消失敗。

unsigned char *State* = GlobalEventControl(unsigned char *Enable*);

啟動或取消所有事件(Event)屏蔽功能。

- ***Enable*** 常數或變數值(0~1)：所有事件屏蔽功能：0: Disable；1: Enable。
- ***State*** 回傳值(0~255)：0: 設定成功；255: 設定失敗。

void GetLibVer(char **Version*);

讀取 Arminno™ Library 版本資訊(20 bytes string data)。

- ***Version*** 指標變數值，回傳 20 byte string 版本資訊。

void GetSysTick(unsigned long *TickValue*);

SysTick 為 Arminno™所提供之一倒數系統時鐘，其時鐘來源速度為 9MHz，倒數數值大小為(0~16777215)3bytes。一旦 Arminno™開始運行 SysTick 即開始不斷循環倒數，此函式是讀取 Arminno™系統時鐘倒數數值，並可做為程式時間參考參數使用。

- **TickValue** 回傳變數值(0~16777215) · SysTick 倒數數值回傳。

unsigned char **State**= SetClkOut(unsigned char **Mode**);

SetClkOut 為 Arminno™所提供之一特定時脈時鐘源方波輸出功能(占用 PC8 腳位) · 可做為其他外部硬體參考使用。

- **Mode** 常數或變數值(0~3) · 時鐘源輸出功能 · 0: Disable · 1: 4.5MHz · 2:0.5MHz · 3:32,768Hz。
- **State** 回傳值(0~255) · 0: 設定成功 · 255: 設定失敗。

```
#include "arminno.h"
#define Led1 PA14
char Version[21];
int main(void)
{
    SetMainClk(1); // 主要時鐘切至內部 8MHz
    SetX32Clk(1);  // 次要時鐘切至內部 32.768 kHz
    SetClkOut(3);
    GetLibVer(Version);
    printf("%s\r\n", Version);
    InitialGpioState(Led1, 1);
    while(1) {
        High(Led1);
        Pause(5000);
        Low(Led1);
        Pause(5000);
    }
}
```

CMDBUS™ 模組物件程式庫應用

有鑑於微控制器可以應用的範圍廣泛，因此利基特別針對微控制器各種不同領域的應用，推出了各種不同的 CMDBUS™ 應用模組，CMDBUS™ 模組內建微控制處理器，可協助處理大量底層複雜及繁瑣之控制程序，例如加速度感測、電子羅盤、溫濕度感測、顏色感測、LCD 顯示、DC 直流馬達驅動、多軸機器人運動.....等等各類應用模組，除了讓使用者可以省下大量開發相關技術的時間之外，更可以藉由各式模組的互相組合發揮過去無法完成的創意與夢想。

為了要讓使用者方便使用 CMDBUS™ 模組，利基整合了所有 CMDBUS™ 模組相關指令，並以 C++ 物件函式庫的方式提供給使用者使用。以下以 LCD2x16A 顯示模組為例：

硬體接線

利用 6PIN 之杜邦排線將 LCD2X16A 模組(Module ID = 31) 之 CMDBUS™ 接頭連接至 Arminno™ 板的 CMDBUS™ 接頭。

軟體物件呼叫

在開發環境主程式 main.cpp 中輸入下面程式：

```
#include "arminno.h"
LCD2X16A myLCD(31);
char Msg[] = {"Hello"};
int main(void)
{
    myLCD.Clear();
    myLCD.BacklightOn(5);
    myLCD.Display(Msg);
}
```

目前已提供的相關 CMDBUS™ 模組如下表：

名稱	說明	相關資料連結
Accelerometer3A	加速度感測模組	Accelerometer 3A 使用手冊
Accelerometer3B	加速度感測模組	Accelerometer 3B 使用手冊
CompassA	方位感測模組	Compass A 使用手冊
CompassB	方位感測模組	Compass B 使用手冊
ColorRGB	顏色感測模組	Color RGB 使用手冊
ThermometerA	溫濕度感測模組	Thermometer A 使用手冊
BarometerA	大氣壓力感測模組	Barometer A 使用手冊
SonarA	距離感測模組	Sonar A 使用手冊
GamepadPS	PS2 搖桿控制模組	Gamepad PS 使用手冊
Joystick2A	2 軸搖桿控制模組	Joystick 2A 使用手冊
Joystick3A	3 軸搖桿控制模組	Joystick 3A 使用手冊
KeypadA	16 Key 鍵盤控制模組	Keypad A 使用手冊
IOExtenderA	I/O 擴充輸入輸出模組	I/O Extender A 使用手冊
LCD2x16A	2x16 行 LCD 顯示模組	LCD 2x16A 使用手冊
LCD4x20A	2x40 行 LCD 顯示模組	LCD 4x20A 使用手冊
MotorRunnerA	1.3A 35V DC 馬達驅動模組	Motor Runner A 使用手冊
MotorRunnerB	雙 650mA 30V DC 馬達驅動模組	Motor Runner B 使用手冊
MotorRunnerC	30A 35V DC 馬達驅動模組	Motor Runner C 使用手冊
MR2x5	雙 5A 35V DC 馬達驅動模組	MR2x5 使用手冊
MR2x30A	雙 30A 35V DC 馬達驅動模組	MR2x30A 使用手冊
ServoRunnerA	16 軸伺服機控制模組	Servo Runner A 使用手冊
ServoRunner8	8 軸伺服機控制模組	Servo Runner 8 使用手冊
TimeKeeperA	萬年曆時間控制模組	Time Keeper A 使用手冊

PlayerA	MP3 音樂播放模組	Player A 使用手冊
RF24G	無線傳輸控制模組	RF24G 使用手冊
UartRunner	UART 傳輸控制模組	UartRunner 使用手冊

ASCII 碼對應表

编码	字符	编码	字符	编码	字符	编码	字符
0	NUL	32	Space	64	@	96	`
1	SOH	33	!	65	A	97	a
2	STX	34	"	66	B	98	b
3	ETX	35	#	67	C	99	c
4	EOT	36	\$	68	D	100	d
5	ENQ	37	%	69	E	101	e
6	ACK	38	&	70	F	102	f
7	BEL	39	'	71	G	103	g
8	BS	40	(	72	H	104	h
9	TAB	41	)	73	I	105	i
10	LF	42	*	74	J	106	j
11	VT	43	+	75	K	107	k
12	FF	44	,	76	L	108	l
13	CR	45	-	77	M	109	m
14	SO	46	.	78	N	110	n
15	SI	47	/	79	O	111	o
16	DLE	48	0	80	P	112	p
17	DC1	49	1	81	Q	113	q
18	DC2	50	2	82	R	114	r
19	DC3	51	3	83	S	115	s
20	DC4	52	4	84	T	116	t
21	NAK	53	5	85	U	117	u
22	SYN	54	6	86	V	118	v
23	ETB	55	7	87	W	119	w
24	CAN	56	8	88	X	120	x
25	EM	57	9	89	Y	121	y
26	SUB	58	:	90	Z	122	z
27	ESC	59	;	91	[	123	{
28	FS	60	<	92	\	124	
29	GS	61	=	93	]	125	}
30	RS	62	>	94	^	126	~
31	US	63	?	95	_	127	DEL