ZBasic Application Note

AN-223 Using Arduino Code in ZBasic Applications

Introduction

The popularity of the Arduino platform has resulted in the availability of a significant amount of embedded application and "library" code for it. Many embedded developers and hobbyists prefer the simplicity of ZBasic rather than learning the Arduino's Wiring-based language (essentially simplified C++) . Fortunately, it is possible to incorporate Arduino procedures, libraries and even entire Arduino applications with relative ease. This application note describes how to accomplish that objective by means of several examples. Note that V4.0.5 or later of the ZBasic compiler is required in order to successfully build ZBasic applications containing Arduino code.

Overview of Arduino Code Structure

The Arduino code base comprises a set of core code and multiple so-called "libraries". (A more precise term would be "source code libraries". Hereinafter, we will use the quoted word "library" or "libraries" to refer to such source code libraries to avoid confusion with the more common use of the words to refer to object code libraries.) The core code, written in C and C++ with a small amount of assembly language, provides the essential services to an embedded application such as pin I/O (set pin state, get pin state), analog-to-digital conversion, PWM and other waveform generation, serial I/O, etc. The Arduino "libraries" provide additional services such as Ethernet connectivity (wired or wireless), LCD interface, stepper motor and servo functions and I2C/SPI functionality. Beyond the official "libraries" provided by the Arduino organization, many Arduino developers have created a "library" for some particular functionality thus allowing others to use it fairly easily. Further, there are many complete Arduino applications available from various sites on the Internet.

We have adapted or re-implemented all of the Arduino core functionality together with most of the official Arduino "libraries" provided with Arduino distributions in order to make the code compatible with ZBasic. This effort required significant changes to the core code to account for the differing assumptions between Arduino and ZBasic (e.g. main clock frequency, pin numbering, timer usage, etc.) and much less extensive changes to the official "library" files. The result is a linkable object library for each supported target device (e.g. ATmega328P, ATmega644P, ATmega1280) together with a set of C/C++ header files describing the available procedures, variables, classes, structures, etc.

The source code for the modified, ZBasic-compatible, Arduino code base and official " libraries" is available in conjunction with this application note. The subsequent portions of the application note will describe 1) how to compile a complete Arduino application for a ZBasic target device, and 2) how to use parts of the Arduino core and/or "library" code in a ZBasic application.

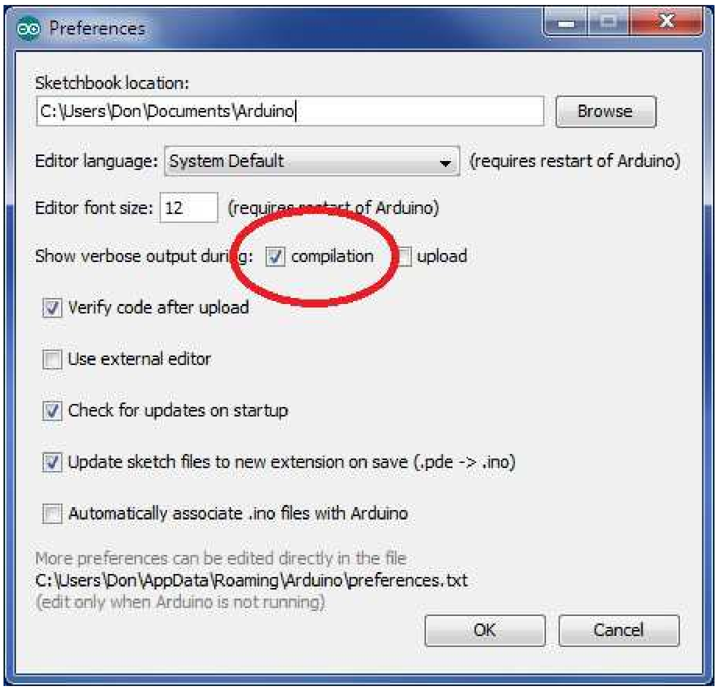
Before proceeding, download and install the ZBasic-Arduino compatibility source code. For the purposes of this application note, we will assume that the entire contents of that archive has been extracted to the directory C:\Projects\Arduino. It is essential that these files are not installed in the "Program Files" directory tree on Windows Vista and later systems because of the "shadowing" feature implemented for such "system" directories.

Compiling a Simple Arduino Application for a ZBasic Target Device

Part of the process required to build an Arduino application for a ZBasic target device involves compiling the application first using the Arduino IDE. If you haven't already done so, download the Arduino installer from the official Arduino webpage (currently, http://arduino.cc/en/main/software). The discussion that follows is based on the Arduino v1.0.3 - some details may be different in newer versions. Versions older than v1.0.2 are not recommended.

Published January 2013

After downloading and installing, launch the Arduino IDE. The next step is to select the File menu and select the Preferences item. In the resulting dialog box, tick the checkbox to enable verbose output during compilation as shown in the image below. Next, load the desired Arduino application, as an example we will choose the "blinky" example application (a.k.a. a "sketch" in Arduino lingo) by selecting the menu item File->Examples->01.Basics->Blink.



Preferences

Sketchbook location:
C:\Users\Don\Documents\Arduino

Editor language: System Default (requires restart of Arduino)

Editor font size: 12 (requires restart of Arduino)

Show verbose output during: [✓] compilation [ ] upload

[✓] Verify code after upload

[ ] Use external editor

[✓] Check for updates on startup

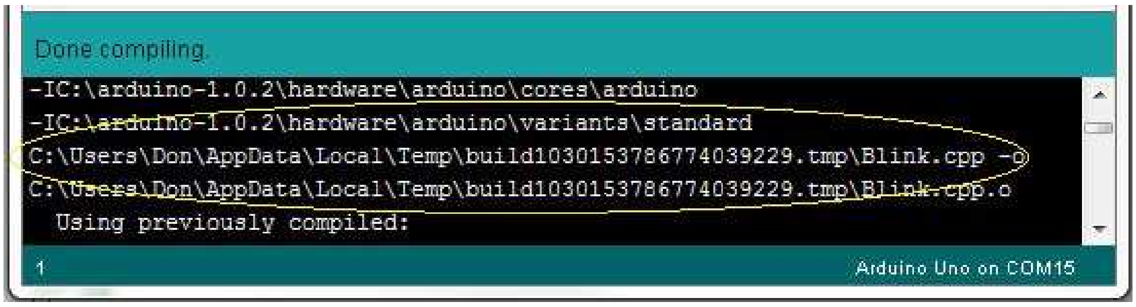
[✓] Update sketch files to new extension on save (.pde -> .ino)

[ ] Automatically associate .ino files with Arduino

More preferences can be edited directly in the file
C:\Users\Don\AppData\Roaming\Arduino\preferences.txt
(edit only when Arduino is not running)

Once the "sketch" is loaded it can be compiled by clicking the "checkmark" button (just below the File menu item) or by selecting "Verify/Compile" from the Sketch menu. Note that at this point it doesn't really matter what Arduino board is selected (Tools->Board) - doing the compile is just a necessary step to get the Arduino IDE to generate the equivalent C++ code comprising the Arduino application.

When the sketch is compiled with the verbose option enabled, a lot of output will appear in the bottom pane of the Arduino IDE window. The important part of this fairly voluminous output is the name of the temporary directory used by the Arduino IDE build process. On a modern Windows platform, the temporary build directory will be a subdirectory of the "user directory" for the user that is logged in. On Win7, for example, the temporary build directory has the form `\Users\<user-name>\AppData\Local\Temp\build<number>.tmp` as shown in the screen capture below.



```
Done compiling.
-IC:\arduino-1.0.2\hardware\arduino\cores\arduino
-IC:\arduino-1.0.2\hardware\arduino\variants\standard
C:\Users\Don\AppData\Local\Temp\build1030153786774039229.tmp\Blink.cpp -o
C:\Users\Don\AppData\Local\Temp\build1030153786774039229.tmp\Blink.cpp.o
  Using previously compiled:
1                                             Arduino Uno on COM15
```

The Arduino build process will create a C++ file named for the sketch that was compiled, in this case Blink.cpp - it is this file that is needed in order to compile the application for a ZBasic target. The next step, then, is to navigate to the temporary directory noted in the Arduino build window and copy the C++ file to a convenient place - as an example, we'll copy it to C:\Projects\ZBasic\Arduino\Blink.cpp. (As a side note, it may be necessary to change your system settings in order to see the AppData directory and its subordinates because that directory usually has the Hidden attribute. On Win7, following the instructions found at http://windows.microsoft.com/en-US/windows7/Show-hidden-files to enable viewing hidden files.)

Now that we have the real C++ source code for the Arduino application, we can proceed to build the application for a ZBasic target. For the purpose of illustration, we will use the ZX-24n as the target device. Launch the ZBasic IDE and create a new project, say ArduinoBlink, using Project->New. The newly created ZBasic program, ArduinoBlink.bas, will initially have the following content:

```
Sub Main()
End Sub
```

Edit the ArduinoBlink.bas file so that it looks like this:

```
Option Arduino "C:/Projects/Arduino"

Sub Main()
   Debug.Print "Running Arduino code"
   Call arduino_setup()
   Do
       Call arduino_loop()
   Loop
End Sub
```

There are several important aspects about this code that should be noted. The first important item is the `Option Arduino` directive. This tells the ZBasic compiler that you are building an application that incorporates Arduino code. Among other things, this makes available two pre-defined "external declarations": the subroutines `arduino_setup()` and `arduino_loop()`. The quoted pathname that follows `Option Arduino` tells the ZBasic compiler where to find the ZBasic-Arduino compatiblility libraries and include files. You'll need to modify this part to match the directory where you installed those files.

The basic structure of all Arduino programs is an initialization routine called `setup()` that is called once and an operation routine called `loop()` that is called repeatedly. If you take a look at the C++ file `Blink.cpp` that we copied, you'll see, in part, the following:

```
int led = 13;

// the setup routine runs once when you press reset:
void setup() {
  // initialize the digital pin as an output.
  pinMode(led, OUTPUT);
}

// the loop routine runs over and over again forever:
void loop() {
  digitalWrite(led, HIGH);   // turn the LED on (HIGH is the voltage level)
  delay(1000);               // wait for a second
  digitalWrite(led, LOW);    // turn the LED off by making the voltage LOW
  delay(1000);               // wait for a second
}
```

The `pinMode()` routine configures a pin as an input or output, similar to the functionality of ZBasic's `PutPin()`. As you might guess, the `digitalWrite()` routine sets the state of an output pin and the `delay()` routine implements a delay of the specified number of milliseconds. The final detail that makes the connection between the ZBasic code and the Arduino C++ code shown above is that the external declarations for `arduino_setup()` and `arduino_loop()` specify that they map to the external subroutines `setup()` and `loop()`, respectively. Creating the aliases in this manner is necessary because `loop` is a ZBasic keyword and thus cannot be used as a subroutine name in ZBasic code. The external declarations could have been implemented directly in the ZBasic code as shown below but since they are often needed when using Arduino code they are supplied automatically by the `Option Arduino` directive.

```
Public Declare Sub arduino_setup() Alias "setup"
Public Declare Sub arduino_loop() Alias "loop"
```

Before attempting to build the application, we must add the Arduino C++ code to the ZBasic project. To do this, select "Open Project File" from the Project menu. Once the file appears, you should see just the first line shown below; add the second line which refers to the Arduino C++ code that was copied earlier from the Arduino IDE's temporary build directory.

```
ArduinoBlink.bas
C:/Projects/ZBasic/Arduino/Blink.cpp
```

After selecting the ZX-24n as the target (Options->Device Options...), press F7 to compile the ZBasic application. If all of the steps described above were followed, the application should compile with no errors. Downloading the code to the ZX-24n should result in the green LED flashing for one second on and one second off.

You may have wondered how it is that the ZX pin connected to the green LED happens to be the one that is toggled by the Arduino `loop()` code. Looking back at the Arduino C++ code above you'll see this line

```
int led = 13;
```

Arduino boards number their digital I/O pins from 0 and up and it is standard to have an LED on digital pin 13. In contrast, in ZBasic one refers to pins either by their physical pin number, e.g. pin 5, or by their port.bit notation, e.g. C.0. Part of Arduino compatibility code is a "digital pin map" that is used to convert Arduino digital pin numbers to the equivalent ZBasic pins. The digital pin map can be found in the Arduino compatibility source file digitalPinMap.c, the portion of which that is relevant to the ZX-24n is (partially) shown below.

```
#elif defined(__AVR_ATmega164P__) || defined(__AVR_ATmega324P__) || \
       defined(__AVR_ATmega644__) || defined(__AVR_ATmega644P__) || \
       defined(__AVR_ATmega1284P__)
   // this data matches the Sanguino pinout
   32,     // number of entries that follow

   // AVR port/pin     Arduino pin
   ZB_PORT(B, 0),      //  0
   ZB_PORT(B, 1),      //  1
   ...
   ZB_PORT(D, 5),      // 13
   ZB_PORT(D, 6),      // 14
```

This data table indicates that Arduino digital pin 13 maps to pin D.5 of the mega644P and similar chips. Other portions of the file describe the pin mapping for other AVR devices that are supported, e.g. mega328P, mega1280, etc. You may recall that pin D.5 on ZX-24n (and similar) devices happens to be connected to the green LED. One slight complication is that on Arduino boards the LED is connected with its anode to the I/O pin so that a logic 1 illuminates the LED. In contrast, on 24-pin ZX devices, the cathode is connected to the I/O pin so that a logic 0 illuminates the LED. In this particular application this difference is unimportant because the LED is alternately turned on and off. In other types of applications where the LED indicates a particular status, the fact that the LED is connected the opposite way will be important. You can deal with this difference by editing the Arduino source code to set the pin to the opposite state.

In some cases, you may want to change the mapping of Arduino pins to ZBasic pins. Such a change can be effected in one of several ways:

- edit the appropriate parts of the digitalPinMap.c file and rebuild the Arduino compatibility library
- copy the digitalPinMap.c file to another directory, edit it, and add it to your ZBasic Project
- add an initialized ProgMem data item to your ZBasic code with the mapping data

The first two methods should be self-explanatory and will not be addressed further. To implement the third method, add something like the following to your ZBasic application:

```
' This is a replacement Arduino pin map table.
Public myArduinoPinMap as ByteVectorData ({
   32  'number of entries that follow
   ' AVR port/pins                                   Arduino pins
   B.0, B.1, B.2, B.3, B.4, B.5, B.6, B.7  '  0- 7
   0,   0,   D.2, D.3, D.4, D.7, D.6, D.5  '  8-15
   C.0, C.1, C.2, C.3, C.4, C.5, C.6, C.7  ' 16-23

   ' analog pins
   A.7, A.6, A.5, A.4, A.3, A.2, A.1, A.0  ' 24-31
}) Attribute(Used, "Alias:digitalPinMap", "PortPinEncoding:1")
```

Several aspects of this construct should be explained further. Firstly, the ZBasic name used , `myArduinoPinMap` above, is immaterial for reasons that will soon become clear. Secondly, the first data item in the table must accurately reflect the number of entries that follow - in this case 32 entries follow, one each for pins 0 to 31. Thirdly, the attributes at the end are important. The `Used` attribute tells the ZBasic compiler not to discard the table even though there appears to be no references to it in the ZBasic program. The `Alias` attribute tells the ZBasic compiler the name to use for the table in the generated source code. This name, `digitalPinMap`, must match the name of the table that the Arduino compatibility code is expecting. Lastly, the `PortPinEncoding` attribute must be set so that the `port.pin` constants are emitted in the encoded form rather than as ZBasic device physical pin numbers.

In the pin map table above, note that the entries for pins 13 and 15 (D.7 and D.5) have been swapped. On a ZX-24n, pin D.5 is connected to the green LED while D.7 is connected to the red LED. With this table in place, compiling and downloading will result in the red LED flashing instead of the green LED.

Building a More Complex Arduino Application for a ZBasic Target Device

The preceding section described a relatively simple Arduino application that contained all of the application code in a single sketch (.ino) file. It is possible (and often advisable) to create Arduino applications that comprise several files including one or more .ino files. For such applications, the Arduino build process will combine all of the .ino files into a single .cpp file in the temporary build directory and the procedure for using that composite file is really no different than the scenario described in the preceding section.

In a more complex Arduino application, in addition to the one or more .ino files there may also be one or more C++ source files and/or header files (.cpp and .h, respectively). In such applications, the Arduino build process combines all of the .ino files into a single .cpp file as described above and then also copies all of the .cpp and .h files to the temporary build directory (this and more details of the Arduino build process are described at http://arduino.cc/en/Hacking/BuildProcess). To build such an application for a ZBasic target it is necessary to copy the generated .cpp file representing the combined .ino files to a convenient directory along with the additional .cpp and .h files.

As an example of an Arduino application comprising .ino, .cpp and .h files, consider the following files:

MyMorse.ino:
```
#include "morse.h"
const uint8_t pin = 13;
Morse morse(pin);

void setup()
{
}

void loop()
{
  morse.dot(); morse.dot(); morse.dot();
  morse.dash(); morse.dash(); morse.dash();
  morse.dot(); morse.dot(); morse.dot();
  delay(3000);
}
```

Morse.h:
```
#if !defined(Morse_h)
#define Morse_h

#include "Arduino.h"

class Morse
{
public:
      Morse(uint8_t pin);
      void dot();
      void dash();
private:
      uint8_t m_pin;    // the pin on which to send Morse code
};
#endif
```

Morse.cpp:
```
#include "Morse.h"

Morse::Morse(uint8_t pin)
{
      m_pin = pin;
      pinMode(pin, OUTPUT);
}

void Morse::dot()
{
      digitalWrite(m_pin, HIGH);
      delay(250);
      digitalWrite(m_pin, LOW);
      delay(250);
}

void Morse::dash()
{
      digitalWrite(m_pin, HIGH);
      delay(1000);
      digitalWrite(m_pin, LOW);
      delay(250);
}
```

To build this Arduino application for a ZBasic target, compile it in the Arduino IDE first to get it to produce the file MyMorse.cpp (in the temporary directory as described earlier) and then copy that file along with Morse.cpp and Morse.h to a convenient directory, say C:/Projects/ZBasic/Arduino. Finally, create a new project in ZBasic named, say, ArduinoMorse.pjt, add the calls to `arduino_setup()` and `arduino_loop()` in the `Main()` as before, and then add the MyMorse.cpp and Morse.cpp files to the project file so that it appears thus:

```
ArduinoMorse.bas
C:/Projects/ZBasic/Arduino/MyMorse.cpp
C:/Projects/ZBasic/Arduino/Morse.cpp
```

As noted earlier, since the LEDs on the ZX-24n chip are connected the opposite way compared to the standard LED on an Arduino, the LED will be off when it should be on and vice versa. This can be corrected in a way that will allow the code to work correctly whether compiled for a ZBasic target or an actual Arduino. Consider this modified Morse.cpp file:

Morse.cpp:
```
#include "Morse.h"

#if defined(ZBASIC_APP)
  #define LED_ON  LOW
  #define LED_OFF HIGH
#else
  #define LED_ON  HIGH
  #define LED_OFF LOW
#endif

Morse::Morse(uint8_t pin)
{
      m_pin = pin;
      digitalWrite(pin, LED_OFF);
      pinMode(pin, OUTPUT);
}

void Morse::dot()
{
      digitalWrite(m_pin, LED_ON);
      delay(250);
      digitalWrite(m_pin, LED_OFF);
      delay(250);
}

void Morse::dash()
{
      digitalWrite(m_pin, LED_ON);
      delay(1000);
      digitalWrite(m_pin, LED_OFF);
      delay(250);
}
```

The identifier `ZBASIC_APP` will be defined whenever code is being compiled for a ZBasic target and not defined when compiling for an Arduino target. Consequently, it can be used as shown above to provide different action in code depending on whether it is compiled for a ZBasic target or an Arduino.

Incorporating Files in Other Directories

The preceding examples have had all C++ and header files in the same directory. While this is simpler, it is sometimes more convenient to segregate C++ files and their related header files in subdirectories. (This is the way that the Arduino "library" files are arranged with each "library" in its own directory, possibly containing subdirectories.) Normally, when the back-end compiler processes C++ files it looks in that same directory for the header files (if the #include directives do not refer to an absolute pathname). To tell the back-end compiler to look in one or more other directories, the `#include_path` directive (or the `--include-path` command line option) can be used.

Continuing with the Morse example code from earlier, the following directory structure might be chosen:

```
C:/Projects/ZBasic/Arduino
+- Morse
|  +- Morse.cpp
|  +- Morse.h
+- MyMorse.cpp
```

To tell the back-end compiler to look in the `Morse` subdirectory for header files, add the following line to the `ArduinoMorse.pjt` project file:

```
--include-path=$(INCPATH);C:/Projects/ZBasic/Arduino/Morse
```

This form, with the `$(INCPATH)` component, results in the specified directory being added to the existing include path. Note that `$(INCPATH)` can appear anywhere in the path list thus allowing complete flexibility in constructing a new include path.

Compiling the application again with the changes for the new configuration should produce the same result as before.

Using Arduino Core Routines Directly in ZBasic Code

The preceding examples showed how to build entire Arduino applications to run on ZBasic targets. It is also possible to incorporate elements of Arduino code in existing or new ZBasic applications. To do this, it is necessary to import the relevant identifiers from the Arduino code. As a simple example, create a new ZBasic project, say MyBlink, and modify the `MyBlink.bas` file as follows:

```
Option Arduino "C:/projects/arduino"
#import "C:/projects/arduino/zbasic_arduino.h"

Const led as Byte = 13  ' N.B.: this is an Arduino pin number
Const OUTPUT as Byte = 1

Sub Main()
   ' initialize the pin as an output
   Call pinMode(led, OUTPUT)

   ' blink the LED
   Do
      Call digitalWrite(led, 1)     ' turn the LED off
      Call delay(1000)              ' wait for a second
      Call digitalWrite(led, 0)     ' turn the LED on
      Call delay(1000)              ' wait for a second
   Loop
End Sub
```

In the code above, the parameter value passed to `pinMode()` that specifies configuration as an output was defined in the ZBasic code - its value was taken from the header file `Arduino.h`. If only a few constant values are needed, duplicating them in the ZBasic code isn't cumbersome but it does introduce the possibility of error. To reduce the chance of error and to facilitate using a large number of Arduino code constants a special form of the #import directive can be used that will also import identifiers that are created in the Arduino headers using the #define preprocessor directive as shown below.

```
#import "C:/projects/arduino/zbasic_arduino.h" "#all-define"
```

With this in place, the definition of OUTPUT in the ZBasic code is no longer necessary. However, the disadvantage of using this form is that it imports hundreds of such identifiers from the Arduino header files and there is a possibility that one or more of those imported identifiers may clash with identifiers defined in the ZBasic code. It is important to note, also, that a constant definition of OUTPUT as shown in the ZBasic Code above quietly overrides any identifier imported from external header files that was created using #define.

Using Arduino-related Routines Directly in ZBasic Code

Using code based on the Arduino core code in ZBasic applications is achieved by simply importing the related header files. This is not much different than importing an arbitrary header file. Consider the ZBasic application below that uses the Morse class defined earlier in the application note. This code assumes that the header file for the Morse class is in the `C:/Projects/ZBasic/Arduino/Morse` directory. Begin by creating a new ZBasic project, say, MyMorse, and modify `MyMorse.bas` to be as shown below.

```
Option Arduino "C:/projects/arduino"
#import "C:/Projects/ZBasic/Arduino/Morse/morse.h"

Const pin as Byte = 13  ' this is an Arduino pin number

Dim myMorse as Morse = _Create(pin)

Sub Main()
   Do
      Call myMorse.dot()
      Call myMorse.dot()
      Call myMorse.dot()
      Call myMorse.dash()
      Call myMorse.dash()
      Call myMorse.dash()
      Call myMorse.dot()
      Call myMorse.dot()
      Call myMorse.dot()
      Call delay(3000)
  Loop
End Sub
```

The only remaining item needed is to add the morse.cpp file to the project file thus:

```
MyMorse.bas
C:/Projects/ZBasic/Arduino/Morse.cpp
```

Compiling this application and downloading it to the target should produce the S-O-S on the LED.

Software

The ZBasic, C, AVR assembly language and associated header files are provided in a .zip archive.

Author

Don Kinzer is the founder and CEO of Elba Corporation. He has extensive experience in both hardware and software aspects of microprocessors, microcontrollers and general-purpose computers. Don can be contacted via email at dkinzer@zbasic.net.