accusoft®

GO AND Barcode Xpress

BarcodeXpress

Barcode Xpress Linux is a C/C++ library compiled for Linux. Many languages have features that allow them to make use of C/C++ libraries. Go, has such a feature and it is called cgo. With cgo, we can write a program entirely in Go that uses cgo to call the necessary Barcode Xpress functions. So, if your codebase is in Go, and you do not want to write a separate application in another language to recognize barcodes, this whitepaper is for you.

Using Barcode Xpress from Go is actually quite easy. Here's a basic example that assumes you have a Barcode Xpress Linux evaluation or development license:

```
package main

/*
#cgo LDFLAGS: -L. -lbarcodexpress -ldl
#include <stdlib.h>
#include "barcodexpress.h"
*/
import "C"
import (
    "fmt"
    "os"
    "unsafe"
)

func main() {

    image := C.CString("/path/to/image.bmp")
    defer C.free(unsafe.Pointer(image))

    // Default settings for barcode reading
    defaultParams := C.BX_DefaultAnalyzeParameters

    // Allocate space for an AnalyzeResult struct for up to 100 barcodes.
    pointerToAnalyzeResults := C.malloc(C.sizeof_struct_tagBX_AnalyzeResult * 100)
    defer C.free(unsafe.Pointer(pointerToAnalyzeResults))

    // Search the image for barcodes and store the results in the struct we allocated space for earlier
    status :=
        C.BX_analyze_file(&defaultParams, image, (*C.struct_tagBX_AnalyzeResult)(
    pointerToAnalyzeResults))

    if status != 0 {
        println("Error:", status)
        os.Exit(1)
    }

    // Create a variable containing the AnalyzeResults struct we allocated earlier.
    analyzeResult := (*C.struct_tagBX_AnalyzeResult)(pointerToAnalyzeResults)
    length := analyzeResult.BarcodeResultsLength

    // The AnalyzeResult struct contains an array of BarcodeResults.
```

```
// Make a big array of BarcodeResults and then make a Go slice for the BarcodeResults from that array.
barcodeResultSlice := (*[1 << 28]C.struct_tagBX_BarcodeResult)
    (unsafe.Pointer(analyzeResult.BarcodeResults)[:length]

// Print out results
for i := range barcodeResultSlice {
    fmt.Printf("# %d\n", i)
    fmt.Println("Type:" + C.GoString(barcodeResultSlice[i].Name))
    fmt.Println("Value:" + C.GoString(barcodeResultSlice[i].Value) + "\n")
}
```

NOW, LET'S GO OVER SOME OF THE CODE ABOVE.

```
/*
#cgo LDFLAGS: -L. -lbarcodexpress -ldl
#include <stdlib.h>
#include "barcodexpress.h"
*/
import "C"
```

The comment before import "C" actually matter. That tells Go what headers, linker options, and compile flags to use.

```
image := C.CString("/path/to/image.bmp")
defer C.free(unsafe.Pointer(image))
```

CString returns a pointer to the start of a char array. So, we must free the memory ourselves. When you're done with the string, simply call C.free. Since we are not done with this string yet, we defer that until the main function exits.

```
defaultParams := C.BX_DefaultAnalyzeParameters
```

Barcode Xpress Linux has a predefined struct called BX_DefaultAnalyzeParameters. This struct defines the settings used for barcode recognition when we call BX_analyze_file. BX_DefaultAnalyzeParameters essentially tells Barcode Xpress to search the entire image for 1D barcodes (Code 39, Code 128, etc).

```
pointerToAnalyzeResults := C.malloc(
    C.sizeof_struct_tagBX_AnalyzeResult * params.MaximumBarcodes)
defer C.free(unsafe.Pointer(ptr))
```

The BX_analyze_file function takes in the address of a BX_AnalyzeResult struct and fills it with information about the barcodes found on the image. So, we need to allocate space for this struct ahead of time. We can get the size of any C type with C.sizeof_Type. Also, because we are calling C.malloc, we must also call C.free.

```
status := C.BX_analyze_file(&defaultParams, image,
    (*C.struct_tagBX_AnalyzeResult)(pointerToAnalyzeResults))
```

Now we call BX_analyze_file. It takes in the address of a BX_AnalyzeParameter struct, a CString for the file path, and the address of a BX_AnalyzeResult struct. It returns a status. A status of 0 means success. Everything else is an error.

```
analyzeResult := (*C.struct_tagBX_AnalyzeResult)(pointerToAnalyzeResults)
length := analyzeResult.BarcodeResultsLength
```

Here we create a variable for our BX_AnalyzeResult struct so that we can access its members easier. We create the length variable here to make the next line of code easier to read.

```
barcodeResultSlice :=
        (*[1 <<28]C.struct_tagBX_BarcodeResult)(unsafe.Pointer(analyzeResult.BarcodeResults))[:length]
```

The BX_AnalyzeResult struct contains a pointer to the first element of an array of BX_BarcodeResult structs. The easiest way to use that array of BX_BarcodeResult structs from Go is with a slice. In the line of code above, we create a very big array of BX_BarcodeResult structs. Then, we create a slice from that array.

```
 for i := range barcodeResultSlice {
         fmt.Printf("# %d\n", i)
         fmt.Println("Type:" + C.GoString(barcodeResultSlice[i].Name))
         fmt.Println("Value:" + C.GoString(barcodeResultSlice[i].Value) + "\n")
 }
```

Now that we have our slice of BX_BarcodeResults, we can iterate through it and print out the results!

While the above code example is simple and easy, it does not show how to customize the BX_AnalyzeParameters. That is important if you want to limit your search to a specific area or only search for certain barcode types. Here's a more complete code example:

```
package main

/*
#cgo LDFLAGS: -L. -lbarcodexpress -ldl
#include <stdlib.h>
#include "barcodexpress.h"
*/
import "C"
import (
    "fmt"
    "os"
    "unsafe"
)

func main() {

    solutionName := C.CString("Accusoft")
    defer C.free(unsafe.Pointer(solutionName))
```

```
oemKey := C.CString("2.0.ReallyLongStringContainingLicenseInformationAndStuff")
defer C.free(unsafe.Pointer(oemKey))

// If you have an evaluation or development license, you do not need to call the following functions.
C.BX_set_solution_name(solutionName)
C.BX_set_solution_key(0, 1, 2, 3)
C.BX_set_oem_license_key(oemKey)

image := C.CString("/path/to/image.bmp")
defer C.free(unsafe.Pointer(image))

// Define BX_BarcodeTypes as constant integers
const (
    BXBarcodeTypeIndustrial2of5       = 1 << iota
    BXBarcodeTypeInterleaved2of5      = 1 << iota
    BXBarcodeTypeIATA2OF5             = 1 << iota
    BXBarcodeTypeDatalogic2of5        = 1 << iota
    BXBarcodeTypeInverted2of5         = 1 << iota
    BXBarcodeTypeBCDMatrix            = 1 << iota
    BXBarcodeTypeMatrix2of5           = 1 << iota
    BXBarcodeTypeCode32               = 1 << iota
    BXBarcodeTypeCode39               = 1 << iota
    BXBarcodeTypeCodabar              = 1 << iota
    BXBarcodeTypeCode93               = 1 << iota
    BXBarcodeTypeCode128              = 1 << iota
    BXBarcodeTypeEAN13                = 1 << iota
    BXBarcodeTypeEAN8                 = 1 << iota
    BXBarcodeTypeUPCA                 = 1 << iota
    BXBarcodeTypeUPCE                 = 1 << iota
    BXBarcodeTypeAdd5                 = 1 << iota
    BXBarcodeTypeAdd2                 = 1 << iota
    BXBarcodeTypeEAN128               = 1 << iota
    BXBarcodeTypePatchCode            = 1 << iota
    BXBarcodeTypePostNet              = 1 << iota
    BXBarcodeTypePDF417               = 1 << iota
    BXBarcodeTypeDataMatrix           = 1 << iota
    BXBarcodeTypeCode39Ext            = 1 << iota
    BXBarcodeTypeCode93Ext            = 1 << iota
    BXBarcodeTypeQRCode               = 1 << iota
    BXBarcodeTypeIntelligentMail      = 1 << iota
    BXBarcodeTypeRoyalPost4State      = 1 << iota
    BXBarcodeTypeAustralianPost4State = 1 << iota
    BXBarcodeTypeAztec                = 1 << iota
    BXBarcodeTypeGS1Databar           = 1 << iota
    BXBarcodeTypeUPU4State            = 1 << iota
    BXBarcodeTypeMicroPDF417          = 1 << iota
    BXBarcodeTypePlanet               = 1 << iota
    BXBarcodeTypeUnknown              = 0
)

params := C.struct_tagBX_AnalyzeParameters{
    ParametersVersion:              1,
```

```
        AppendCheckSum:                  false,
        Area:                            C.struct_tagBX_Rectangle{C.long(0), C.long(0), C.long(0), C.long(0)},
        // Barcode types can be OR'd together
        BarcodeTypes:                    BXBarcodeTypePDF417 | BXBarcodeTypeAztec,
        IncludeControlCharacters:        true,
        MaximumBarcodes:                 100,
        MinimumBarcodeSize:              0,
        Orientation:                     3, // BX_Orientation_HorizghontalVerticalDiagonal
        ReturnPossibleBarcodes:          false,
        AustralianPostDecodeType:        3, // BX_AustralianPostDecodeType_NoCustomDecode
        RoyalMailVariableLengthDecoding: false,
        ScanDistance:                    5,
        GrayscaleProcessing:             false,
        AdditionalReadingPass:           false,
    }

    // Allocate space for an AnalyzeResult struct for up to MaximumBarcodes
    pointerToAnalyzeResults := C.malloc(C.sizeof_struct_tagBX_AnalyzeResult * params.MaximumBarcodes)
    defer C.free(unsafe.Pointer(ptr))

    // Search the image for barcodes and store the results in the struct we allocated space for earlier
    status := C.BX_analyze_file(&params, image, (*C.struct_tagBX_AnalyzeResult)(pointerToAnalyzeResults))
    if status != 0 {
        println("Error:", status)
        os.Exit(1)
    }

    // Create a variable containing the AnalyzeResults struct we allocated earlier.
    analyzeResult := (*C.struct_tagBX_AnalyzeResult)(pointerToAnalyzeResults)
    length := analyzeResult.BarcodeResultsLength

    // The AnalyzeResult struct contains an array of BarcodeResults.
    // Make a big array of BarcodeResults and then make a Go slice for the BarcodeResults from that array.
    barcodeResultSlice := (*[1 << 28]C.struct_tagBX_BarcodeResult)
        (unsafe.Pointer(analyzeResult.BarcodeResults))[:length]

    for i := range barcodeResultSlice {
        fmt.Printf("# %d\n", i)
        fmt.Println("Type:" + C.GoString(barcodeResultSlice[i].Name))
        fmt.Println("Value:" + C.GoString(barcodeResultSlice[i].Value) + "\n")
    }
}
```

NOW, LET'S GO OVER THE NEW THINGS ADDED TO OUR CODE.

```
    solutionName := C.CString("Accusoft")
    defer C.free(unsafe.Pointer(solutionName))

    oemKey := C.CString("2.0.ReallyLongStringContainingLicenseInformationAndStuff")
    defer C.free(unsafe.Pointer(oemKey))

    // If you have an evaluation or development license, you do not need to call the following functions.
    C.BX_set_solution_name(solutionName)
    C.BX_set_solution_key(0, 1, 2, 3)
    C.BX_set_oem_license_key(oemKey)
```

Here are some licensing functions you'll need to call if you're using a deployment or OEM license.

```
// Define BX_BarcodeTypes as constant integers
    const (
        BXBarcodeTypeIndustrial2of5       = 1 << iota
        BXBarcodeTypeInterleaved2of5      = 1 << iota
        BXBarcodeTypeIATA2OF5             = 1 << iota
        BXBarcodeTypeDatalogic2of5        = 1 << iota
        BXBarcodeTypeInverted2of5         = 1 << iota
        BXBarcodeTypeBCDMatrix            = 1 << iota
        BXBarcodeTypeMatrix2of5           = 1 << iota
        BXBarcodeTypeCode32               = 1 << iota
        BXBarcodeTypeCode39               = 1 << iota
        BXBarcodeTypeCodabar              = 1 << iota
        BXBarcodeTypeCode93               = 1 << iota
        BXBarcodeTypeCode128              = 1 << iota
        BXBarcodeTypeEAN13                = 1 << iota
        BXBarcodeTypeEAN8                 = 1 << iota
        BXBarcodeTypeUPCA                 = 1 << iota
        BXBarcodeTypeUPCE                 = 1 << iota
        BXBarcodeTypeAdd5                 = 1 << iota
        BXBarcodeTypeAdd2                 = 1 << iota
        BXBarcodeTypeEAN128               = 1 << iota
        BXBarcodeTypePatchCode            = 1 << iota
        BXBarcodeTypePostNet              = 1 << iota
        BXBarcodeTypePDF417               = 1 << iota
        BXBarcodeTypeDataMatrix           = 1 << iota
        BXBarcodeTypeCode39Ext            = 1 << iota
        BXBarcodeTypeCode93Ext            = 1 << iota
        BXBarcodeTypeQRCode               = 1 << iota
        BXBarcodeTypeIntelligentMail      = 1 << iota
        BXBarcodeTypeRoyalPost4State      = 1 << iota
        BXBarcodeTypeAustralianPost4State = 1 << iota
        BXBarcodeTypeAztec                = 1 << iota
        BXBarcodeTypeGS1Databar           = 1 << iota
        BXBarcodeTypeUPU4State            = 1 << iota
        BXBarcodeTypeMicroPDF417          = 1 << iota
        BXBarcodeTypePlanet               = 1 << iota
        BXBarcodeTypeUnknown              = 0
    )
```

Barcode Xpress uses an enum to define the various barcode types. Each value in our enum is a power of 2. Here we make use of the "iota" feature of the Go language. It resets to 0 every time the "const" keyword is used and increments for every const specification.

```
params := C.struct_tagBX_AnalyzeParameters{
        ParametersVersion:               1,
        AppendCheckSum:                  false,
        Area:                            C.struct_tagBX_Rectangle{C.long(0), C.long(0), C.long(0), C.long(0)},
        // Barcode types can be OR'd together
        BarcodeTypes:                    BXBarcodeTypePDF417 | BXBarcodeTypeAztec,
        IncludeControlCharacters:        true,
        MaximumBarcodes:                 100,
        MinimumBarcodeSize:              0,
        Orientation:                     3, // BX_Orientation_HorizontalVerticalDiagonal
        ReturnPossibleBarcodes:          false,
        AustralianPostDecodeType:        3, // BX_AustralianPostDecodeType_NoCustomDecode
        RoyalMailVariableLengthDecoding: false,
        ScanDistance:                    5,
        GrayscaleProcessing:             false,
        AdditionalReadingPass:           false,
    }
```

BarcodeXpress

Customizing BX_AnalyzeParameters is probably the most important part of the new code we added in this example. With this struct, we can greatly alter the way Barcode Xpress searches for barcodes.

It is important to note that BarcodeTypes is an integer, and each BX_BarcodeType is a power of 2. So, in order to recognize multiple barcode types, we simply OR the barcode types we want together.

If you like, you can OR every barcode type together to search for all barcodes we support. For Orientation and AustralianPostDecodeType, I've simply passed in integers. You can create your own const and define them like I did for BX_BarcodeTypes.

So, there you have it! Using Barcode Xpress from Go is pretty straightforward. Now that we've reviewed the more complex steps, you can get started on your own. For more information on Barcode Xpress, visit the developer resources tab on our product page.

www.accusoft.com