# MapGenerator High Level Design

Table of Contents

[MapGenerator High Level Design 1](#_Toc240085088)

[Versions 2](#_Toc240085089)

[OverView 3](#_Toc240085090)

[Encoding 4](#_Toc240085091)

[Decoding 5](#_Toc240085092)

[Inheritance Table 6](#_Toc240085093)

[Objects Table 7](#_Toc240085094)

[Classes 7](#_Toc240085095)

[Libraries 7](#_Toc240085096)

## Versions

|  |  |  |  |
| --- | --- | --- | --- |
| *Version* | *Main Change* | *Author* | *Date* |
| 0.01 | First Copy | Roi | 03.09.09 |
| 0.02 | Objects Table Added | Roi | 07.09.09 |
|  |  |  |  |

## OverView

The following documents describes in high level the method in which Script language is turned into Barcode Buffer (the actual creating of the barcode later is immediately, using the Barcode Buffer and a third party BarcodeEncoder (such as QR code encoder) which takes buffer and creates an image).

The Barcode Creating Process is as follows:

Note: Current documentation only specifies Data Encoding, since the Scripting and Compiling doesn't exist yet.

The Map Generating process is as follows:

Note: Current documentation relates only to Data Decoding.

## Encoding

On the left there is a graphical diagram of what the BarcodeEncoder receives as an input and what it returns as an output. On the right is a recipe for using the BarcodeEncoder.

Header

Header

Header

BarcodeEncoder

Header

Header

Item

Buffer ready to become barcode

The following is example for a code which encodes a parking Item (performs steps 2 and 4):

SStructureShape StructureShape(SStructureShape::None, SStructureShape::SShape());

SMultiplicity Multiplicity((m\_NumberOfObjects > 1), m\_NumberOfObjects);

CItemStructure Structure;

Structure.Encode(StructureShape, Multiplicity);

IItem \*ParkingItem = NULL;

switch ((EMapObjectType)ObjectTypeIndex)

{

case NORMAL\_PARKING:

ParkingItem = new CParkingItem;

((CParkingItem \*)ParkingItem)->Encode(Perpendicular, Regular, false, Degrees0, Structure, false);

break;

case PARALLEL\_PARKING:

SPeriodicBetweenPoles Periodic(12, SPeriodicBetweenPoles::Circular);

ParkingItem = new CParkingItem;

((CParkingItem \*)ParkingItem)->Encode(Parallel, Handicap, true, Degrees180, Structure, true, &Periodic);

break;

};

## Decoding

In order to decode, you simply call DecodeBuffer function of the CBarcodeDecoder, and the function returns a list of headers and items, as graphed bellow:

Header

Header

Header

BarcodeDecoder

Header

Header

Item

Buffer ready to become a parking map

No recipe for using the barcode decoder is attached, as there is simply one call. All the interesting algorithms is done within. Therefore attached bellow is a diagram demonstrating the way the BarcodeDecoder works:

The real interesting part is how the internal specific decoder decodes the information (speficially the CBarcodeParkingMapDecoder):

* This is not yet explained, as I haven't implemented it yet.

The following is an example code for decoding a parking item:

int UsedBits = 0;

ParkingItem->Decode(ParkingItem->GetBitBuffer(), UsedBits);

## Inheritance Table

The following table describes the abstract object types, and the inherited implemented objects:

These are the currently implemented objects. More should be created in the future.

## Objects Table

### Classes

1. CBarcodeDecoder – The main class which is responsible for decoding a barcode buffer. It uses all kinds of other decoders to decode a specific kind of barcode (ParkingMap, Text, Url, etc.).
2. CBarcodeEncoder – The main class which is responsible for encoding a barcode buffer.
3. CBarcodeParkingMapDecoder – This class is responsible for decoding all ParkingMap related data. The CBarcodeDecoder class is more general and can theoretically decode barcode buffers other than Parking Maps, such as pure texts, url links and so on. Of course the main operation (from our point of view) is the parking map decoding, which is why technically the CBarcodeDecoder will be pretty "empty" and all the logic will be here.
4. CBasicItem - This class inherits from IItem and represents a basic item in the map (e.g. elevator, staircase, WC, etc.)
5. CBit – This class implements a bit (Normally C allows programs to access data by byte. This extends it to bits)
6. CBitPointer - This class utilize the CBit described above, and creates a BitPointer object. Using the CBitPointer we can hold bit buffers.
7. CComplexItem – another type of item (inherits from IItem). The CComplexItem allows the in-barcode definition of repeating structures and sub-structures (in a recursive fashion).
8. CGeneralHeader – This is the header of the whole barcode (we will use it to: 1. Recognize it is our barcode. 2. Get version number, and barcode type). There will be further header(s) according to the barcode type specified in the general header. This header also inherits from IHeader
9. IHeader – an abstract class representing all headers.
10. IItem – an abstract class representing all items.
11. IItemDecoder – OBSELETE
12. CItemsFactory – Factory for creating IItems according to enum.
13. CItemStructure – A sub Item. Doesn't stand for itself, but only as a part of a "real" item, such as the ParkingItem.
14. CParkingDecoder – OBSELETE
15. CParkingItem – An item which inherits from IITem. Represnts a parking space(s). it has ItemStucture within which allows the parking item to holds many parking space in various orientations.
16. CParkingMapHeader – Inheirts from the IHeader, and implements the header of the Parking Map Barcode.

### Libraries

1. BitLib.h – This library holds several usefull BitBuffer manipulation routines.
2. ItemHelper.h – This library holds many MACROs regarding the definition of bit-types, and macros which helps copying data to and from a bitbuffer.