# MapGenerator High Level Design

Table of Contents

[MapGenerator High Level Design 1](#_Toc240816185)

[Versions 2](#_Toc240816186)

[OverView 3](#_Toc240816187)

[Encoding 4](#_Toc240816188)

[Decoding 5](#_Toc240816189)

[Inheritance Table 6](#_Toc240816190)

[Items Syntax Definition 7](#_Toc240816191)

[Intorduction 7](#_Toc240816192)

[BasicItem 7](#_Toc240816193)

[ComplexItem 7](#_Toc240816194)

[PositionItem 7](#_Toc240816195)

[PavementItem 8](#_Toc240816196)

[Objects Table 9](#_Toc240816197)

[Classes 9](#_Toc240816198)

[Libraries 9](#_Toc240816199)

## Versions

|  |  |  |  |
| --- | --- | --- | --- |
| *Version* | *Main Change* | *Author* | *Date* |
| 0.01 | First Copy | Roi | 03.09.09 |
| 0.02 | Objects Table Added | Roi | 07.09.09 |
| 0.03 | Items Syntax Definition Added | Roi | 10.09.09 |
| 0.04 | BasicItem, PositionItem, PavementItem definition added.  Graphics added to flowcharts.  Small typos fixed. | Felix | 15.09.09 |
| 0.05 | PavementItem and BasicItem updated to include new Items and functionality. | Felix | 19.10.09 |

## 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**

**Barcode Encoder**

**Buffer ready to become barcode**

Header

Header

**Item**

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

**Item**

Header

Header

**Header**

**Barcode Decoder**

**Buffer ready to become barcode**

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.

## Items Syntax Definition

### Intorduction

Each letter represents a bit. Big letters represent decision making values (such as Booleans and Enums), and small letters represent values (such as numerical values and textual values). Numerical values can also be defined using round brackets and number of bits within. Strong brackets indicate this value is optional (and depends on prior bits).

### BasicItem

1110tttt(b)(xxxx xx)(xxxx xxxx xx(xxxx xx)) (yyyy yy)(yyyy yyyy yy(yyyy yy))

tttt – BasicItem type.

Currently defined basic items: Elevator, Staircase\_H, Staircase\_V, WC, CarGate\_Right, CarGate\_Left, CarGate\_Up, CarGate\_Down, PayingMachine, Door, Sign, Column\_Sqr, Column\_Cir, Road.

All items except: CarGate\_Right, CarGate\_Left, CarGate\_Down, Door and Road can be scaled.

b – 'IsScaled' bit used to enable scaling.

(xxxx xx), (yyyy yy) – When 'IsScaled' bit set to '1', two 6 bit fields are included to represent X and Y scaling coordinates.

CarGate items are not scalable but the length and the direction of the gate can be controled by using the CarGate\_Up item and setting 'IsScaled' to '1'. X and Y coordinates in this case represent the CarGate edge point.

(xxxx xxxx xx(xxxx xx)), (yyyy yyyy yy(yyyy yy)) - Road item includes two 10 or 16 bit fields that representsX and Y coordinates of an oposite corner for a rectanglular road section. Road is always rectanglular and current carriage location is used as a first corner. 'IsScaled' bit is used to select ether 10 or 16 bit coordinates size.

'Door' item orientation is set to be parallel to the orientation of a boundary on which the 'Door' is placed.

### ComplexItem

1111uuuuu(nnnnn)MvMhVH(R)(sssssstttttt) (sssssstttttt)

uuuuu – UID

nnnnn – number of objects (only for first time definition)

Mh – horizontal mirroring

Mv – horizontal mirroring

V – Duplicate Vertically

V – Duplicate Horizontally

R – Is duplication part of the definition (only for first time definition)

ssssss – space between duplicates (6 bits)

tttttt – Times to duplicate (6 bits)

### PositionItem

10HV[xxxxxxxx][yyyyyyyy]

H – Horizontal Jump

V – Vertical Jump

xxxxxxxx – horizontal jump length (8 bit)

yyyyyyyy – vertical jump length (8 bit)

When setting value of Horizontal+Vertical bits to 00 this means a **CARRIAGE RETURN.**

The position is **FORWARD and RELATIVE** (which is the most frequent one). It allows small relative jumps downwards and rightwards(in order to use Absolute Jumps or Backward jumps (up and\or left) another item type should be used).

Position units will be in physical decimeters (0.1m) regardless of parking garage measurements. This will allow easier faster movements on smaller parking places. We currently find the minimum size of 0.1m and maximum size of 6.55km as unlimiting.

The relative position is **cyclic** meaning that if the carrier is at the top, moving up will cause it to arrive from the bottom upwards (the same distance as specified). Same applies for all 4 directions.

### PavementItem

110WW [ZZ(C)][(6 bit)] S [ZZ(C)(6+2**S** bit)(6+2**S** bit)][ZZ(C)(6+2**S** bit)(6+2**S** bit)][ZZ(C)(6+2**S** bit)(6+2**S** bit)][ZZ(C)(6+2**S** bit)(6+2**S** bit)] [TT][GGGG]

Pavement will be based on vertexes and curve type describing the shape of the pavement.

The Pavement Item will behave differently when adjacent to parking compared to non-adjacent, while adjacent to parking can be either before or after the parking spaces (for a vertical parking spaces either directly above or below, and for horizontal parking spaces either directly to the left or the right). Currently, adjacent also requires the Pavement command to be also adjacent to the parking command in the script itself (either before or after) for simplifying the Decoder work. This constraint can be removed in the future.

S - SizeOfVertex / SideOfTriangle

    S

    0 - Not adjacent to parking and using 8 bit per vertex coordinate

    1 - Not adjacent to parking and using 6 bit per vertex coordinate

    0 - Adjacent to parking and triangle edge is on the same side as carriage

    1 - Adjacent to parking and triangle edge is on the opposite side from carriage

WW = ShapeID (see the detail below).

When not adjacent to parking, texture type is added to the item: 110….TT[GGGG]

TT - texture type

    00 - pavement

    01 - building

    10 - no parking

    11 - solid fill

If 'solid fill' is chosen, brightness must be specified:

GGGG - brightness

    0000 - solid white

    ...

    ...

    1000 - 50% grey

    ...

    ...

    1111 - solid black

In addition to the header, each line has a curvature parameter:

ZZ - edge curvature type:

00 - linear

01 - Convex

10 - Concave

11 – Rectangular

C – IsCircular. When Convex or Concave curvature is chosen additional bit is added to distinguish between oval and circular shape types.

The ShapeID value is determent automatically from the following algorithm:

**if IsAdjacentToParking == true:**

**=======================**

1. **ShapeID==0 if ShapeType == Wall.** There will be no vertices, and no length to vertice (no ShortenVertexCoordinate & no m\_SizeOrSide !! )
2. **ShapeID==1 if ShapeType == Triangle/Right\_Triangle/Rectangle:** There will be one more vertex, the (ShortenVertexCoordinate & m\_SizeOrSide) vertex. ShortenVertexCoordinate controls the distance to the vertex (perpendicular to the adjacent parking), two sides will be linear, and the curvature of the hypotenuse will be determined by the m\_SpecialVertexCurvature.   
   If the curvature type is rectangle, then the pavement will be rectangular.
3. **ShapeID==2 if ShapeType == Quadrangle**. One side is adjacent to the parking. One is perpendicular to parking (with ShortenVertexCoordinate as the distance, SpecialVertexCurvature for the curvature and m\_SizeOrSide for selecting which side is perpendicular), and two other free vertices with curvature.
4. **ShapeID==3 if ShapeType == Pentagon**. Analogous to the Quadrangle but in this case three vertices with curvature are included.

**if IsAdjacentToParking == false:**

**=======================**

1. **ShapeID==0 if ShapeType == Wall/RightTriangle/Rectangle**. For all three scenarios, there will be one more vertex with curvature, aside for the SpecialVertexCurvature.
   * If the SpecialVertexCurvature==VertexCurvature==Linear\Convex\Concave, then there will be a simple wall, its length and orientation simply dictated by the coordinate of the vertex (its shape determined by the curvature).
   * If the SpecialVertexCurvature==VertexCurvature==Rectangular, then there will be a rectangular, its diagonal length and orientation simply dictated by the coordinate of the vertex.
   * If the SpecialVertexCurvature!=VertexCurvature then it is a right triangle. Since there are two Curvature fields, their order matters and dictates whether it is a left-triangle or right-triangle. Notice that the curvature which has the Rectangle is the one with the right angle. If there is no such, the curvature which has the Concave (rounded right angle) is the right angle. If there is no such, then the linear curvature is NOT the right angle.
2. **ShapeID==0 if ShapeType == Triangle** Two vertices determine the next two sides and the curvature to them. the SpecialVertexCurvature determines the last curvature back to origin. Drawing goes clockwise.
3. **ShapeID==0 if ShapeType == Quadrangle**: three vertices determine the next three sides and the curvature to them. The SpecialVertexCurvature determines the last curvature back to origin. Drawing goes clockwise.
4. **ShapeID==0 if ShapeType == Pentagon**: four vertices determine the next four sides and the curvature to them. The SpecialVertexCurvature determines the last curvature back to origin. Drawing goes clockwise.

## 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 – OBSOLETE
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 – OBSOLETE
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 – Inherits from the IHeader, and implements the header of the Parking Map Barcode.

### Libraries

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