Populating the VELO Connectivity Database

Robert John Shade
Summer Student 2006

September 22, 2006
1 Introduction

The Vertex Locator (referred to as VELO from now on) is a sub-detector of the LHCb. It will accurately measure vertices of b-hadron decays, providing, amongst other things, vital data for the level 1 trigger system.

The VELO must be able to detect and measure such decay vertices in three dimensional space and thus has both radial (r) sensors and angle (ϕ) sensors. It contains over 20 modules on each detector half, with each module containing both an r-sensor and a ϕ-sensor, mounted back to back. Each sensor is read out by an electronic read-out system consisting of multiple read-out sensors, and numerous cables and electronics boards. This arrangement of devices (cables, boards etc.) must be described in the VELO connectivity database, and it was my task to investigate how this should be done. This work is described in section 3, and section 4.

A database visualisation program, cdbVis will be used to view the data in a convenient manner, and add and modify additional data. The use of this program will also be discussed in section 5.
2 Definitions

**Device:** Any physical component to be stored in the database. This includes kapton cables, beetle read-out chips, silicon sensors etc. An exhaustive list can be found in the device naming section.

**Macroscopic/Microscopic device:** A microscopic device is a device which is contained within another device. An example would be a driver mezzanine, which is located within a repeater board. In this case the driver mezzanine is the microscopic component and the repeater board is the macroscopic component. Any device which is not contained within another device is referred to as a macroscopic device.

**Port:** In the database a device must have ports associated with it. These ports are input/output points on the device. For example, a uni-directional cable will have an input port at one end and an output port at the other.

**Link:** A connection between two devices. This is different to modelling a cable (such as a kapton) as a device. For example, the short kapton cables connect to the long kapton cables by means of links. In reality the cables are just connected to each other by their ends. A link is just a way of representing devices that are connected to each other.

**Device type:** General description of a device, containing properties that are common to all devices of that type. For example, every short kapton cable in the VELO has 1 input, 1 output, and is a certain colour. The device type describes all this, whereas a specific device is an instance of this type, with additional details (serial number, name, location etc.)
3 Existing database

An existing database - the LHCb configuration database - already contained tables to store the components and the links or connections between them. These was already being partially used by the muon and DAQ systems, but not the VELO.

3.1 Database structure

A quick overview of the database table structure.

$LHCB\_LG\_DEVICE\_TYPES$ contains data which describes all the available device types in the connectivity database.

Table $LHCB\_LG\_DEVICES$ stores each device as a separate row. Each row is uniquely identifiable by use of the $SERIALNB$ (serial number), the $DEVICENAME$ (device name), and the $DEVICEID$ fields. $DEVICENAME$ must be unique, but other than that there are no constraints on naming (although a sensible option will be discussed later) - the important fields for device identification are $SERIALNB$ and $LOCATION$. The serial number uniquely identifies a physical device, and the location tells you where it is currently physically located. The naming conventions will be described in a later section.

Device ports are stored in the $LHCB\_PORT\_PROPERTIES$ table. Each row details a port’s properties, such as $PORT\_TYPE$, $IP\_ADDRESS$ etc. The field $DEVICEID$ refers directly to the corresponding $DEVICEID$ in $LHCB\_LG\_DEVICES$, hence specifying which device this port is a part of.

$LHCB\_MACROSCOPIC\_CONNECTIVITY$ stores details of all the links in the system. Each row in the table stores the connected ports in the columns $PORTIDFROM$ and $PORTIDTO$.

$LHCB\_LG\_CPNTS$ stores details regarding microscopic devices. This is very similar to the $LHCB\_LG\_DEVICES$ table and needs no further discussion.

Similarly, $LHCB\_MICROSCOPIC\_CONNECTIVITY$ is the microscopic equiva-
lent of LHCb\textunderscore MACROSCOPIC\textunderscore CONNECTIVITY. However, microscopic connectivity works slightly differently. A microscopic link can be from the motherboard to a microscopic component, from a microscopic component to the motherboard, or an internal connection from one port of the motherboard to another. Each row in the table stores the component ID which begins the link (CPNTIDFROM), component ID which ends the link (CPNTIDTO), and the corresponding ports (PORTIDFROM and PORTIDTO respectively). If the start of the link is the motherboard then CPNTIDFROM has the value -1.0, and the PORTID of the motherboard port is given in PORTIDFROM. The CPNTIDFROM is given a proper value if indeed the starting device is a microscopic component. The reverse situation is true if the link is going from a microscopic component to the motherboard.


4 VELO naming conventions

The various devices associated with a single silicon sensor can be seen in Figure 1 (page 10). The device types created to represent these in the database will be described below.

4.1 Device type names

VELO_HYBRID: Hybrid is a macroscopic device that houses the 16 beetle read-out chips (known as VELO_FE_CHIPs in the database)

VELO_FE_CHIP: There are 16 of these devices per sensor, and each one reads out 128 strips of the sensor

VELO_SHORT_KAPTON: Each short kapton takes signals from 8 beetles. In Figure 1 the right hand short kapton will take signals from beetles 0-7 inclusive, and the right hand kapton will take signals from beetles 8-15 inclusive

VELO_LONG_KAPTON: The 4 long kaptons continue to carry the data signals - it can be seen in Figure 3 exactly which beetle signals each long kapton carries

VELO_FEEDTHROUGH_FLANGE: The feedthrough flange is a panel which separates the secondary vacuum of the VELO and the air of the pit. Each feedthrough flange has 20 inputs and 20 outputs, and thus can cope with up to 5 front end circuits (4 long kaptons enter and 4 interconnects per front end circuit)

VELO_INTERCONNECT: The interconnects are very short pieces of cable which simply connect the feedthrough flange to the repeater board

VELO_REPEATER: The repeater board contains multiple microscopic components (described below). It is responsible for amplifying the signals, and keeping in contact with the control system

VELO_LV_MEZZANINE: The LV mezzanine provides the power connection for the whole repeater board.
VELO DRIVER MEZZANINE: The 4 driver mezzanines amplify the signals coming from the 4 interconnects. Figure 2 shows the signal paths from interconnects, through the driver mezzanines and out of the repeater board.

VELO ECS MEZZANINE: The ECS (Electronic Control System) mezzanine is the connection between the front end electronics and the VELO control system.

VELO TEMPERATURE: This is one of three boards not shown in Figure 1. It takes two temperature readings from the front end circuit and processes these. There are 6 temperature boards in total; 3 per VELO half.

VELO CONTROL: The control board takes the signal from the ECS mezzanine. Again this board is not shown in Figure 1. Each control board deals with signals from 6 repeater boards.

VELO TELL1: This is the read out board which takes the actual data signals (after amplification by the driver mezzanines) and does some initial data processing. There is one TELL1 board for each repeater board.

VELO ARX MEZZANINE: Each TELL1 board contains 4 ARX mezzanines. These are therefore microscopic devices.
4.2 Device naming convention

Initially the plan was to come up with a device naming system similar to devices already in the configuration database, and link this unique name to the device’s physical location. Later, this was discovered to be impractical, as a device may well end up being moved between multiple locations. To solve this problem we decided to use generic names for the devices, and use the LOCATION field to specify physical location.

4.2.1 Sensible values for DEVICENAME field

Although the DEVICENAME (or CPNTNAME for microscopic devices) field for a device can contain any value, as long as it is unique, it wouldn’t hurt to name devices sensibly. The suggestion is to take the device type, VELO_{LONG}APTON for example, and add a number on the end as the DEVICENAME. This way, when browsing in the tree view in cdbVis (see section 5), or reading the text file ‘reality check’ output (see section 5.2), it is instantly clear what device type we are dealing with.

Every time a new device is created, the name will be identical to the last device of that type, but with the final number incremented by 1. cdbVis already contains code which tries to predict the best naming scheme for new devices, and follows this convention.

4.3 Location naming convention

After discussion with Paula Collins, the following naming conventions were suggested for the location field of macroscopic and microscopic devices (LOCATION in LHC\_LG\_DEVICES and in LHC\_LG\_CPNTS respectively). We will look at each device type in turn.

The nomenclature adopted in M. Ferro-Luzzi’s document ‘Definition of PU and VELO detector layout’ (EDMS Nr. 719303 v1, 14 June 2006) will be followed as closely as possible.

Each module is named VLXXY, where XX is a two digit number (ascending from 01 at the most upstream module to 25 at the most downstream module), and Y is the
Figure 1: One silicon sensor and associated devices
Figure 2: Repeater board connectivity
Figure 3: Two VELO modules
letter R or the letter L depending on which half of the VELO the module is located. e.g. VL09L is the 9th module on the left hand side of the VELO.

NOTE: This numbering is related to the slots, and some slots don’t contain modules. 00-16 all contain modules, but then 17,18,20,21 are empty. This is why the last modules will be numbered VL25R and VL25L instead of VL21R and VL21L.

Each module has two sensors attached, as mentioned above. The 4 kaptons leaving each sensor will leave via the top of the VELO (if the sensor is on the upstream side) or the bottom of the VELO (if the sensor is on the downstream side). Interconnects are labeled according to the associated slot. The labels take the form VLXXY/Cm where VLXXY is the slot label, C is either ‘t’ for top or ‘b’ for bottom, and m is an integer [1..4] which designates which one of the 4 cable interconnects we are referring to.

The distinction between top and bottom is important, as the cables are leaving the VELO in entirely opposite directions, as seen in Figure 3.

Module: As described above, each module is named in accordance with existing documentation. This means it takes the form of VLXXY where XX is the slot number, and Y is ‘L’ or ‘R’, designating left or right detector halves.

• VL03L
• VL12R
• VL09R
• etc.

Hybrid: A hybrid’s location should consist of the module name in which it is present, an underscore ‘_’, the sensor type ‘R’ or ‘PHI’, another underscore ‘_’ and finally ‘UPSTREAM’ or ‘DOWNSTREAM’, which indicates which side of the module it is attached to. Now this location scheme is slightly redundant as the sensor type (R or PHI) could be deduced from the module name and UPSTREAM or DOWNSTREAM, and vice versa. However it was felt that to save time it was worth including both pieces of information in the location field.
• VL16L_R_UPSTREAM
• VL05_PHI_DOWNSTREAM
• etc.

**Front end chips (beetles):** The beetles have identical locations to the hybrid in which they are located, with an additional number on the end, from 0 to 15 inclusive. These numbers specify the location of the beetle, starting at 0 at the top right of the sensor and moving clockwise up to 15. See Figure 1.

• VL16L_R_UPSTREAM_6
• VL05_PHI_DOWNSTREAM_14
• etc.

**Short kapton:** Short kapton location’s begin with the location of the hybrid they are attached to, followed by an underscore ‘_’, and ending with a number 0, 1, 2, or 3. This numbering scheme reflects the clockwise numbering system of the beetles, and is shown in Figure 1.

• VL01L_R_UPSTREAM_0
• VL01L_R_UPSTREAM_1
• VL01L_R_UPSTREAM_2
• VL01L_R_UPSTREAM_3
• etc.

**Long kapton:** Long kapton locations are identical to the locations of the short kaptons they are connected to.

• VL01L_R_UPSTREAM_0
• VL01L_R_UPSTREAM_1
Feedthrough flange: There are 10 feedthrough flanges for each half of the detector; 5 at the top, 5 at the bottom. The location naming scheme reflects this. The location begins with a number, 0 to 4. This indicates which position it is in along the detector (0 is most upstream feedthrough flange, 4 is furthest downstream). This number is followed by either TOP or BOTTOM, and then LEFT or RIGHT.

- 0_TOP_LEFT
- 3_BOTTOM_RIGHT
- etc.

Interconnect: The interconnect locations are identical to the location of the feedthrough flange they are connected to, but with an additional number on the end. This number specifies which output port of the feedthrough flange the interconnect is attached to, and can range from 0 to 19 inclusive (each feedthrough flange has 20 ports). (NEED DIAGRAM OF FEEDTHROUGH FLANGE PORTS)

- 4_TOP_LEFT_0
- 4_TOP_LEFT_1
- etc.

Repeater board: Repeater boards have locations identical to the feedthrough flange to which they are connected, but with an additional number on the end. This number specifies which row of the feedthrough flange it is connected to (via the interconnects). (NEED DIAGRAM OF FEEDTHROUGH FLANGE PORTS)

- 4_TOP_LEFT_0
• 4_TOP_LEFT_1
• etc.

**Mezzanines:** The mezzanines (LV, ECS, and Driver) have locations that first of all specify the repeater board on which they are mounted, followed by the location on the board.

• 4_TOP_LEFT_0_J18
• 4_TOP_LEFT_0_J19
• etc.

**TELL1, Temperature and Control boards:** All three boards will have locations that specify which rack and crate they are in.

• RACK_3_CRATE_5
• RACK_9_CRATE_1
• etc.

### 4.4 Serial numbers

Accurate serial numbers must be entered into the 'serialnb' field of each device. Without this data we would have no 'link to reality' and no way of matching up devices in the database with the physical devices in the VELO.

### 4.5 Spare devices

To allow differentiation between spare devices, and devices installed in the VELO, the location of the spare device must be prefaced with the word *SPARE*. This is so that cdbVis can work out if a device is a spare or not, and filter the list accordingly.

For example, suppose we have a spare Driver mezzanine, and it is being stored in storage cupboard 7, shelf 4. In the database the location field of this device should be
something along the lines of *SPARE\_CUPBOARD\_7\_SHELF\_4*. Of course the words cupboard and shelf may be replaced with whatever would be a sensible description of the spare part’s location.
5 cdbVis and confDB library

To work out which devices are connected by a link, for example, takes a lot of work if done by hand. We would need to look up the link’s LINKID and find the two connected ports. Once we have these we can look up the PORTIDs in the LHCB_PORT_PROPERTIES table. We will now have the DEVICEIDs of the two connected devices, which we can look up in the LHCB_LG_DEVICES table to get the device details.

Luckily such tedious work isn’t required because of a combination of the cdbVis visualisation program and the confDB database interface. The confDB libraries contain all the methods required to insert, view and modify device, port, and link details in the database.

cdbVis is a Python program which allows the user to view the contents of the connectivity database graphically, showing the devices and connections between them and so on. By making calls to the confDB libraries, it also allows the user to create new device types, devices, ports etc. and also modify existing devices.

Figure 4: cdbVis displaying some devices and links
5.1 Downloading and installing cdbVis

cdbVis can be downloaded from the lhcb-online web pages:

http://lhcb-online.web.cern.ch/lhcb-online/configurationdb/cdbVis.htm

A users guide which explains in detail how to install cdbVis for use with either Linux or Windows is available on the same pages:

http://lhcb-online.web.cern.ch/lhcb-online/configurationdb/Documents/cdbVis/cdbvisnote2006028.pdf

5.2 Outputting a text file for ’reality check’

An important and useful feature that I implemented is the ability to output the contents of the database as a text file. This file describes each device’s name, location, and serial number. By printing this file and checking the details against reality we can ensure that the data in the database is up to date and accurate.

The details are automatically written to a text file called device−details.txt in the cdbVis installation directory. The result can be seen in Figure 5.

This is done using the mainWindow.ExportText function found in cdbVis.py. It opens a pop-up window for the user to enter a filename, checks the extension is .txt and corrects if necessary, and then writes all details to the file. The details are retrieved from each device object in turn using the GetName(), GetLocation(), and GetSerial() methods.

5.3 Filtering spare devices

The tree view on the left hand side of the main cdbVis window displays available sub-systems, then available device-types, and then available devices. However, until now there was no way of differentiating between spare devices (devices currently in storage) and devices ’in place’.
Figure 5: Text file output of device names and locations
There is now an option to select from three different filter options. These are 'All', which displays all devices, 'Spares only', which displays only spare devices, and 'In place only', which only displays devices currently installed in the VELO. The drop down selection is shown in Figure 6.

The file `selectPanel.py` contains all the code dealing with the construction of the left hand panel of the main cdbVis window. This file was modified to contain an additional drop down box (Figure 6). When a choice is made, the tree is collapsed, refreshed, and opened up again to the sub-system the user was viewing. This is done in the `selectWindow.OnRestrictedSparePartChoice` method.

It would be nice if the tree opened up fully, to the lowest (device) level that the user was viewing when they made their choice, and this should be considered as a possible enhancement in the future.
The \texttt{selectWindow.UpdateDeviceType} method is called whenever a device type is double clicked, and displays the available devices to the user. It is in this method that we do the filtering of devices based on spare/in place. Each device retrieved from the database is checked to see if the word 'SPARE' appears in its location field, and if so it is marked as a spare part. It is then easy to filter based on the combination of this and the value of the drop-down selection box.

5.4 Viewing microscopic devices

When a device is viewed in cdbVis there should be a way of identifying that it contains microscopic devices or not. The simplest solution that I came up with was to draw the device with a solid red border.

This does not check in the database for microscopic components belonging to that device (although since there is \texttt{GetMicroscopicDevices} function for each device object this would be easy to do), instead it refers to a list of device types which are known to contain microscopic devices. This list is called \texttt{hasMicroscopicComponents} and is defined in \texttt{cdbVisCore.py} on line 73.

Line 4715 in \texttt{visWindow.py} checks if the current device is of a type that is in the list, and if so draws a solid red border around it.

Additionally there is a new toolbar button (top toolbar, on the far right), which is enabled when a device containing microscopic devices is selected. This has no functionality at the moment other than printing the microscopic devices present to the standard output.

5.5 Mass insertion of data to connectivity database

A Python script has been written that will write a full ‘slice’ of the VELO into the connectivity database. A slice is defined as all the devices that are associated with one sensor. This script is located on the Twiki pages - see section 7 for details.
This file could be used as the basis of mass insertion of lots of VELO devices. By copying and pasting the code and editing it to include correct serial numbers and locations and then running it, you will be able to enter a batch of devices a lot more easily than creating each one in cdbVis.
6 Further work to be done on cdbVis

The major feature missing from cdbVis that will prevent painless usage by the VELO team is the ability to create/modify/view microscopic devices. I will discuss the situation and my ideas on how to implement this feature - I have unfortunately run out of time to do it myself.

6.1 Accessing microscopic Device data from other classes

The main problem seems to be that the methods used to retrieve and display device data, specifically those located in visWindow.py are too tightly bound to the attributes of macroscopic devices. The Device class in objectclasses.py contains methods for querying a device object - initialising a device, getting and setting attributes.

The device attributes are stored in a list inside each device object. This list is called deviceInfo. A Python list works like a extendable, associative array. This means that you add items to it in the form key:value, and retrieve the data by making calls similar to val = deviceInfo[key].

Now when a device object is created, and updated from the database using the Update method, this deviceInfo list is filled with entries corresponding to each data field in the database. However, the fields for macroscopic devices are, in many cases, different to the fields for microscopic devices. For example, a macroscopic device has an attribute ‘node’, which a microscopic component doesn’t, and a microscopic component has an attribute ‘motherboardid’ which a macroscopic device does not.

The problems begin when we try to view a microscopic device in cdbVis. Because the code has been written entirely with macroscopic devices in mind, it isn’t a simple matter of creating a new Device object and filling it with microscopic device details.

A major problem seems to be that there are multiple ways of getting data from a Device object. Every attribute that a Device can have has both a SetXXX and a GetXXX method inside the Device class (where XXX is of course the name of the
attribute, such as Name or Location). This is a nice way to provide an abstract interface with an object, without the client needing to know about the underlying data representation.

Unfortunately these methods aren’t the only way to access the data. If the client calls the method `device.GetObjectInfo` then they are returned the entire `deviceInfo` list, containing all the data held by the device object. This has only become a problem since the viewing of microscopic devices was considered. If we are only dealing with macroscopic devices then it matters little whether we call `device.Get_Name()` or `device.GetObjectInfo()['devicename']`. However, since microscopic devices do not have a ‘devicename’ attribute (their name is stored as ‘cpntname’), we have a bit of a problem when the second method of retrieving a name. When this is tried, Python understandably throws a KeyError exception.

The class `visWindow` in the file `visWindow.py` is responsible for creating the main frame in cdbVis and drawing objects in it when they are selected. In an ideal world all the interaction between `visWindow` and `Device` objects would be through the `Get` and `Set` methods provided, and never by retrieving the entire data list. If this were the case then the alterations to allow for microscopic devices would be minimal; the `GetName()` method would, for instance, check if it was part of a microscopic or macroscopic object, and then return either ‘cpntname’ or ‘devicename’ respectively.

Alas, a lot of such calls are done using the second method mentioned above, and thus cause us problems. Instead something similar to the following must be done everywhere where such a call is made:

```python
if deviceInfo[IS_MICRO]:
    name = deviceInfo['cpntname']
else:
    name = deviceInfo[DEV_NAME]

node = self.FindByName(name)
```

25
This is ugly, and prone to errors: Every single place in the code where the deviceInfo list is used rather than the Get/Set methods must be changed to something similar, and any number of these could be missed.

Instead, the best idea is to get rid of all the calls to objectclasses.Device.getObjectInfo, and replace them with appropriate Get/Set calls. The next section will give numerous examples to explain this more clearly.

### 6.1.1 Some examples to make this description clearer

Some examples of the problem described above will now be given, with filenames and line numbers.

The Device class starts on line 1060 of objectclasses.py. Devices (both microscopic and macroscopic) are instances of this class. Every device has a number of attributes: DEV_PORTS_IN, DEV_NAME, DEV_COLOUR etc. Starting on line 1120, every attribute has both a Get and a Set method. This is demonstrated in Code Listing 2.

#### Code listing 2 Get and Set methods for DEV_LOCATION attribute.
```python
def GetLocation(self):
    return self.__deviceInfo[DEV_LOCATION]
...

def SetLocation(self,location):
    Validation.ValidateString("Location",location,200,True)
    self.__deviceInfo[DEV_LOCATION] = location
```

As you can see, these attributes are stored in the _deviceInfo list, and should be accessed by calls to the appropriate Get/Set methods. However, as described previously, quite often the calls are to the getObjectInfo method (line 1276 of objectclasses.py, and then use the returned list to get the desired attributes. An example is shown in Code Listing 3.

This example is using calls to a link object, not a device, but it demonstrates the point nicely. The call linkobj.GetObjectInfo(True) returns the whole of the attribute list, which is clearly inefficient. A much more sensible way of doing things would be as
Inefficient method for retrieving attributes `DEV_SWITCH_FROM` and `DEV_SWITCH_TO`

```
link = linkobj.GetObjectInfo(True)
node = link[DEV_SWITCH_FROM]
nodes[node] = node
node = link[DEV_SWITCH_TO]
nodes[node] = node
```

shown in Code Listing 4.

Efficient method for retrieving attributes `DEV_SWITCH_FROM` and `DEV_SWITCH_TO`

```
node = linkobj.GetDevSwitchFrom()
nodes[node] = node

node = linkobj.GetDevSwitchTo()
nodes[node] = node
```

This reduces the number of method calls to the link object by 1, and makes our job a lot easier. If this style of coding is adopted throughout the cdbVis codebase wherever a device’s attributes are required then it will make adding microscopic functionality a lot simpler.

To find places where these changes need to be made, search the code for the string `GetObjectInfo`, and then examine and fix the code that follows these calls.

Now, in the `objectclasses.Device` class some changes will need to be made to the Get/Set methods. These will need to include a check to see if the device is a microscopic device (`.deviceInfo[IS_MICRO]` should tell you that), and then returning the correct information. This is shown in Code Listing 5.

```
Example of change to GetLocation method in `objectclasses.Device` class

def GetLocation(self):
    location = None
    if self.__deviceInfo[IS_MICRO]:
        location = self.__deviceInfo['cpntlocation']
    else:
        location = self.__deviceInfo['location']
    return location
```
This will work because although both microscopic and macroscopic devices have a
location field, the names of the fields are different. Now when cdbVis wants to know a
device location it just calls Device.GetLocation() and without caring if it is microscopic
or macroscopic, will get the correct location returned.

6.2 Creating/Modifying microscopic devices

There already exists a sensible method for creating and modifying macroscopic devices in
cdbVis, accessible via the Create -¿ Device and Modify -¿ Device menus respectively. To
allow microscopic devices to be created and modified using the same interface, an option
(such as a checkbox 'Microscopic Device’) must be available in ‘Create Device’ pop-up
window to specify that the device is to be microscopic, and have text boxes available to
enter the data that is unique to microscopic devices.

It would make sense for the macroscopic only fields to be greyed out (inactive) when
the 'Microscopic Device' checkbox is ticked, and for microscopic only fields to be greyed
out when the checkbox is not ticked. This should be fairly straightforward to implement;
the existing code for Creating and Modifying devices is found in CreateDevice.py. A
callback method must be written that toggles the correct input fields between active and
inactive states when the 'Microscopic Device’ checkbox is ticked, and sets a local variable
specifying that we are creating a microscopic device.

Additionally we must ensure that the correct command is sent to the database when we
decide to store the object - this must be set in the method objectclasses.Device.PrintSave
near line 2060, and in objectclasses.Device.Save around line 2135.

6.3 Severing links when location changes

When moving a device to a new location, it is necessary to delete all links currently
attached to the device, and then create links to the devices to which it is now attached.
This could be a time consuming task, and would be greatly simplified by having an option
to ‘swap’ two devices, and have the link details automatically changed.

6.4 Changing a motherboard device location

When a device containing microscopic devices (such as a repeater board) is moved location, presumably the microscopic devices will tend to move with it. For this reason it would be useful to have the option to move a motherboard and all of its components to a new location, and to update the locations of all devices concerned.

6.5 Viewing microscopic components

As discussed in the previous section, when a device containing microscopic components is drawn on screen it is drawn with a red border, and the ’Microscopic view’ button on the toolbar is enabled when it is selected. However, this button has no functionality yet.

The proposal is that when the button is clicked a small window will pop-up containing all the microscopic devices belonging to that macroscopic device. You will be able to view how they are connected, and modify them in the standard way.

The callback function of the ’Microscopic view’ button is called MicroscopicView and is located on line 661 of cdbVis.py. Code must be added here to display this pop-up window. The method for opening a pop-up window when a button is clicked can be found in the method CreateDeviceClick on line 3911 of the same file. This should hopefully simplify the coding of this new method.

Additionally a couple of ideas were suggested to make it more visible that a device contains microscopic devices. In addition to the red border, we could display a message in the status bar at the bottom of the cdbVis window, and/or provide a ‘tooltip’, i.e. when the user hovers the mouse over a device, a message window will display if it contains microscopic devices.
7 Related documents

This is a list of all related documentation, including links to documents referenced in this report, and links to additional material which may be of use.

7.1 Naming conventions

'Definition of PU and VELO detector layout' - M. Ferro-Luzzi (EDMS Nr. 719303 v1, 14 June 2006)

Located on EDMS:


7.2 Full size images

Full size versions of images shown in this document (Figures 3, ??, and 2) are located on the LHCb Twiki pages.

https://uimon.cern.ch/twiki/bin/view/LHCb/VeloConnectivity

As well as these PNG images, the SVG (Scalable Vector Graphics) source files are included, so that they may updated easily at a future date. A recommended piece of software for editing SVG files is 'Inkscape', a free, open-source editor. Inkscape can be downloaded from: http://www.inkscape.org/

7.3 Python code for generating VELO ’slice’

A file containing Python code to enter an entire 'slice' of the VELO into the connectivity database is also located on the LHCb Twiki pages, specifically here:

https://uimon.cern.ch/twiki/pub/LHCb/VeloConnectivity/insert_full_slice.py.txt

This file only correctly describes the connectivity of one slice (defined as the front end
electronics associated with one silicon sensor), and does not contain real data. This file could be used as a basis for entering accurate data into the database, if cdbVis is deemed too cumbersome (or is unavailable on your system).

7.4 Report source

This document was written in Latex, and the source file is available on the LHCb Twiki pages. This may be of use if parts of this file are to be used at a later date, and needs to be re-formatted. You will find the source file here:

https://uimon.cern.ch/twiki/pub/LHCb/VeloConnectivity/robert_shade_report.tex
8 Acknowledgements and contact information

I would like to thank Eric van Herwijnen for supervising this project, Paula Collins for providing feedback on my process, and Lana Abadie for help with the configuration database.

Please contact Eric van Herwijnen (Eric.van.Herwijnen@cern.ch) or Paula Collins (Paula.Collins@cern.ch) with any queries regarding this report.