BRUNEL

LHCb Reconstruction Program

User Guide

Corresponding to Brunel version v9r1

Version: 9.1
Issue: 0
Date: 20 March 2002

European Laboratory for Particle Physics
Laboratoire Européen pour la Physique des Particules
CH-1211 Genève 23 - Suisse

Document Control Sheet

Document	**Title:**	BRUNEL User Guide		
	Version:	9.1		
	Issue:	0		
	Edition:	0		
	ID:	[Document ID]		
	Status:			
	Created:	25 May 2000		
	Date:	20 March 2002		
	Access: :			
	Keywords:			
Tools	**DTP System:**	Adobe FrameMaker	**Version:**	6.0
	Layout Template:	Software Documentation Layout Templates	**Version:**	V1 - 15 January 1999
	Content Template:	--	**Version:**	--
Authorship	**Coordinator:**	M.Cattaneo		
	Written by:	M.Cattaneo		

Document Status Sheet

Title:	BRUNEL User Guide		
ID:	[Document ID]		
Version	**Issue**	**Date**	**Reason for change**
5	3	13 September 2001	Modifications for v5r2, v6r1, v7r1
8	0	19 October 2001	Updated for Brunel v8
9	0	20 January 2002	Updated for Brunel v9r0
9	1	20 March 2002	Updated for Brunel v9r1

Table of Contents

Document Control Sheet 2
Document Status Sheet 2

Table of Contents 3

Chapter 1
Introduction 7
1.1 Purpose and structure of this document 7
1.2 Package structure 7
1.3 What does "Brunel" mean? 8
1.4 Editor's note 8

Chapter 2
Brunel program structure 9
2.1 *Introduction* 9
2.2 *Brunel phases* 9
2.2.1 Initialisation 9
2.2.2 Reconstruction phases and sequences 10
2.2.3 Finalisation 11
2.3 Program configuration 11
2.3.1 Instantiation of Brunel phases 11
2.3.2 Instantiation of Brunel sequences 11
2.3.3 Instantiation of sub-system algorithms 12

Chapter 3
Current Implementation 13
3.1 Introduction 13
3.2 Supported versions and compatibility with input data 13
3.3 Detector Description 14
3.4 Event Data Model 15
3.4.1 PileUp 15
3.4.2 SpillOver 15
3.5 Random numbers 16
3.6 Digi Phase 16
3.6.1 VELO 17
3.6.2 Inner Tracker 17
3.6.3 Outer Tracker 17
3.6.4 RICH 17
3.6.5 CALOrimeters 17
3.6.6 MUON 17
3.7 Trigger phase 18

3.7.1 Technical Proposal (TP) trigger algorithms . . . 18
3.7.2 L0 trigger . . . 18
3.8 Reco Phase . . . 18
3.8.1 Inner Tracker . . . 18
3.8.2 Outer Tracker . . . 18
3.8.3 Velo Tracking . . . 18
3.8.4 Forward tracking . . . 19
3.8.5 Upstream tracking and track fit . . . 19
3.8.6 RICH . . . 19
3.8.7 CALOrimeters . . . 19
3.9 Final Fit phase . . . 19
3.10 Moni Phase . . . 19
3.10.1 ITMCHitsMonitor, ITDigitsMonitor . . . 19
3.10.2 OTDigitChecker . . . 20
3.10.3 TrMonitor, TrCreat, TrFitIn . . . 20
3.10.4 L0CaloMonit . . . 20
3.10.5 ITDigitChecker . . . 20
3.10.6 FwtAnalyse . . . 20
3.10.7 SpdMonit, PrsMonit, EcalMonit, HcalMonit . . . 20
3.11 Output data . . . 21

Chapter 4
Customising and Running Brunel . . . 23
4.1 Introduction . . . 23
4.2 Modifying the run time behaviour . . . 23
4.2.1 Database selection . . . 24
4.2.2 Defining input data . . . 24
4.2.3 Defining output data . . . 25
4.2.4 Modifying the printing behaviour . . . 26
4.2.5 Monitoring options . . . 27
4.2.6 Enabling static execution . . . 29
4.2.7 Additional job options . . . 29
4.3 Adding user code . . . 30
4.4 Building and running the job . . . 30
4.4.1 Running the default version . . . 31
4.4.2 Running the default version with modified job options . . . 31
4.4.3 Running the default version with modified requirements . . . 32
4.4.4 Building a modified version . . . 32
4.5 Problem reporting and resolution . . . 32
4.5.1 Known Problems . . . 32
4.5.2 Getting help . . . 33
4.5.3 Reporting problems . . . 33

Appendix A

References . . . 35

Chapter 1
Introduction

1.1 Purpose and structure of this document

This document is a user guide and reference manual for the LHCb reconstruction program, Brunel. It should be used in conjuction with documentation available on the web, at *http://cern.ch/lhcb-comp/Reconstruction/* and should be useful both to users wishing to run the program, and to programmers wishing to add functionality. Chapter 2 describes the structure of the program. The current functionality is described in Chapter 3. Chapter 4 describes how users can modify the program's functionality and its run time behaviour.

Brunel is based on the Gaudi software framework [1], and uses CMT [2] for code management. This guide assumes some familiarity with both of these tools. Please refer to the corresponding documentation for details on these topics.

This document does not describe the physics algorithms or the data model. A compilation of notes discussing LHCb reconstruction algorithms and the LHCb data model is available on the Web [3],[4].

1.2 Package structure

Brunel is implemented as a CMT package, in the "`Rec`" package group, with the following subdirectory structure:

- `doc` release notes
- `job` example job for Linux
- `options` default job options and SICB data cards
- `cmt` CMT `requirements` file. The Brunel version is defined at compile time via a compiler switch defined in this `requirements` file.

— `Visual` Visual Studio Workspace

This package is used exclusively to build and execute the application. The code for the Brunel framework is found in two other packages in the `Rec` package group:

— `BrunelKernel` Component library for the Brunel framework

— `BrunelSICB` Interfaces classes and subroutines to SICB Fortran code. This package consists of a library (`BrunelSICBLib`, built from sources in the `src/Lib` sub-directory) and of a number of Fortran routines (in `src/Objs`) which must be linked explicitly into the Brunel executable to replace routines with the same name in other SICB packages.

1.3 What does "Brunel" mean?

All LHCb data processing applications are based on a framework which enforces the GAUDI architecture. Antoni Gaudi [5] was a Catalan architect who greatly influenced the development of Barcelona around the beginning of the nineteenth century. For the reconstruction program we decided to use the name of an engineer. Isambard Kingdom Brunel [6] was a British engineer who greatly contributed to the industrial revolution in the first half of the eighteenth century.

1.4 Editor’s note

This document is a snapshot of the Brunel software at the time of the release of version v9r1. We have made every effort to ensure that the information it contains is correct, but in the event of any discrepancies between this document and information published on the Web, the latter should be regarded as correct, since it is maintained between releases and, in the case of code documentation, it is automatically generated from the code.

We encourage our readers to provide feedback about the structure, contents and correctness of this document and of other Gaudi documentation. Please send your comments to the editor, *Marco.Cattaneo@cern.ch*

Chapter 2
Brunel program structure

2.1 Introduction

The Brunel reconstruction program is built on the Gaudi framework, and provides mechanisms for sequencing reconstruction algorithms within this framework. Algorithms executed in Brunel have access to all the services currently implemented in Gaudi, and to all data in the Gaudi data stores, as documented in the Gaudi user guide [1].

Reconstruction in Brunel is executed in a number *phases*, each of which can contain *sequences* of sub-detector algorithms. The instantiation of phases and the sequencing of algorithms within the phases are driven by *job options*.

2.2 Brunel phases

2.2.1 Initialisation

Brunel is initialised in the `BrunelInitialisation` algorithm. The SICB specific initialisation is performed in the `BrunelInitSicb` algorithm. These Gaudi algorithms are where all initialisations which are independent of BrunelPhase are performed. These can be global program initialisations (in the `initialize()` method), or event by event initialisations (in the `execute()` method). Note that initialisations specific to a given BrunelPhase should not be performed here.

In the current version, `BrunelInitialisation` explicitly creates the Event Data Store directory tree required by the reconstruction algorithms. In a future version (when the new event model is adopted) this will be created automatically by Gaudi.

2.2.2 Reconstruction phases and sequences

This where the meat of the reconstruction program lies. Actual phases are derived from the `BrunelPhase` base class. Each `BrunelPhase` should be independent of other `BrunelPhases`: it should be possible to run only one phase, providing of course that event input data in the appropriate format exists[1]. All initializations and finalizations specific to the phase should be performed inside the phase.

Each phase consists of a number of Gaudi *Sequences*, typically one per sub-detector, which execute a set of reconstruction algorithms in a predefined order.

The following `BrunelPhases` are currently instantiated:

- **`BrunelDigi`** is where simulated RAW Hits are converted into DIGItisings. The output of this phase has the same format as real RAW data coming from the detector[2]. Obviously this phase would not be present when reconstructing real data, and could be moved to the simulation program when reconstructing simulated data. Note that this implies some discipline when designing the DIGItised data model, in particular for what concerns links to Monte Carlo truth information.
- **`BrunelTrigger`** is where the LHCb trigger decision is applied. The input event data are DIGItisings. The output are also DIGItisings, with the addition of the trigger decision information.
- **`BrunelReco`** is where the first pass reconstruction is carried out. By first pass we mean that the reconstruction algorithms in this phase rely only on DIGItisings and do not require input from the reconstruction of other subdetectors. This restriction can be somewhat relaxed by ensuring that subdetectors are reconstructed in a specific order: those that only require input from the DIGItisings are processed first, those that require input from the reconstruction of other sub-detectors are processed after those sub-detectors.
- **`BrunelFinalFit`** is the second pass reconstruction, to allow for processing which requires input from the reconstruction of several subdetectors.
- **`BrunelMoni`** is a phase intended purely for monitoring, which could be switched off during a large production. Monitoring histograms should, wherever possible, be filled by algorithms that execute in this phase. This phase is discussed in more detail in section 4.2.5.1

Note that additional phases could easily be implemented if further reconstruction passes are required.

1. This is not entirely true in the current version of the reconstruction program, due to the underlying calls to SICBDST routines, which do not have this structure.
2. This is not entirely true in the current version of the reconstruction program, due to the underlying use of the SICB data model, which does not have this structure

2.2.3 Finalisation

Brunel is finalised in the **`BrunelFinalisation`** algorithm. This Gaudi algorithm is where all the finalisations which are independent of `BrunelPhase` are performed. These can be global program finalizations (in the `finalize()` method), or event by event finalisations (in the `execute()` method). Note that finalisations specific to a given `BrunelPhase` should not be performed here.

2.3 Program configuration

As with other programs based on Gaudi, Brunel is configured through job options. Several job options files are distributed with Brunel, in the `/options` sub-directory. Here we describe some features of the main job options file, `Brunel.opts`, which sets up the standard configuration of Brunel and should not normally be changed by the user. Other job options files that can be modified to customise the run time behaviour of Brunel are described in Chapter 4.

In addition, Brunel needs to know which database version to use. This is described in Section 4.2.1.

2.3.1 Instantiation of Brunel phases

Brunel Phases are Gaudi top Algorithms. They are therefore instantiated using the standard Gaudi job option `ApplicationMgr.TopAlg`. Listing 2.1 shows the default value of this option for the current implementation. Note the different phases in lines 3 to 6, which are different instances of the `class BrunelPhase` and will be executed in the order shown

Listing 2.1 Brunel Top Algorithms as defined in `$BRUNELOPTS/Brunel.opts` job options file

```
1:  ApplicationMgr.TopAlg = { "BrunelSicbInit",
2:                            "BrunelInitialisation/BrunelInit",
3:                            "BrunelPhase/BrunelDigi",
4:                            "BrunelPhase/BrunelTrigger",
5:                            "BrunelPhase/BrunelReco",
6:                            "BrunelPhase/BrunelFinalFit",
7:                            "BrunelFinalisation/BrunelFinish" };
```

2.3.2 Instantiation of Brunel sequences

Reconstruction code should be executed inside Brunel Phases. Each Brunel Phase instantiates a Gaudi *Sequence* for each sub-detector or sub-system participating in that phase. The instance name of the Sequence follows a specific convention: it is composed of the Phase name (e.g. `BrunelDigi`) followed by an abbreviated sub-system name (e.g. `MUON`), followed by the string "`Seq`" (e.g. `BrunelDigiMUONSeq`). These Sequences are intended to be the phase specific steering algorithms of the sub-systems: the subsystem reconstruction algorithms have to be declared as *members* of the Sequence.

Listing 2.2 shows how the different sub-system sequences are instantiated within the existing Brunel phases, in version v9r1 of Brunel.

Listing 2.2 Brunel Sequences as defined in `$BRUNELOPTS/Brunel.opts` job options file

```
BrunelDigi.DetectorList = { "VELO" , "IT", "OT" , "RICH" , "CALO", "MUON" };
BrunelTrigger.DetectorList = { "TRIGGER" };
BrunelReco.DetectorList = { "OT", "IT", "Tr" , "TRACK", "RICH" , "CALO" };
BrunelFinalFit.DetectorList = { "TRAC" };
```

2.3.3 Instantiation of sub-system algorithms

Within each Sequence, the sub-systems are able to instantiate any number of algorithms by simply adding the appropriate job option. For example, to instantiate the `ECALSignal` and `HCALSignal` instances of the `CaloSignalAlgorithm` in the `BrunelDigiCALOSeq` sequence (and to execute ECAL before HCAL), one would add the following job option:

```
BrunelDigiCALOSeq.Members += { "CaloSignalAlgorithm/EcalSignal" ,
                               "CaloSignalAlgorithm/HcalSignal" };
```

The advantage of this system is that it is easily extendable and modifiable. To add a new phase, or a new sub-detector, or a new algorithm (or to change any of their names), it is sufficient to make the necessary changes to the job options. No changes are necessary to the Brunel steering code.

In order to further simplify the maintenance of Brunel and of the sub-system code, a convention has been adopted [7] whereby the sub-systems provide a file called `Brunel.opts` in the `/options` sub-directory of the sub-system algorithms package. This file contains the sequence member declarations for the sub-system algorithms, and any other options required to run the algorithms in Brunel. Typically these would be the additional libraries to be loaded at run-time (`ApplicationMgr.DLLs` job option) and further include files for the algorithm specific options. An example is shown in Listing 2.3

Listing 2.3 The `$TRALGORITHMSROOT/options/Brunel.opts` file

```
ApplicationMgr.DLLs   += {"VSicbCnv", "VeloEvent"};
BrunelRecoTrSeq.Members +={"TrTrueTracksCreator",
                           "TrTracksCreator/TrCreat",
                           "TrFitInitializer/TrFitIn",
                           "TrEventTracksFitter/trFit"};

#include "$TRALGORITHMSROOT/options/trailFit.opts"
```

This and other similar files are then included in the standard Brunel job options file as shown in Listing 2.4.

Listing 2.4 Inclusion of the Tracking algorithms in the `$BRUNELOPTS/Brunel.opts` job options file

```
#include "$TRALGORITHMSROOT/options/Brunel.opts"
```


Chapter 3
Current Implementation

3.1 Introduction

The currently supported versions of Brunel are "hybrid" versions, containing Fortran code from the old SICBDST packages and new C++ code. Communication between the various FORTRAN algorithms is done, as in SICBDST, via COMMON blocks, in particular the ZEBRA common block. The C++ algorithms have access to all the Gaudi services, and in particular the Gaudi data stores. Data is exchanged between the Fortran and C++ worlds by means of *converters* which convert data from SICB banks to Transient Event Data objects and vice versa.

The FORTRAN algorithms are controlled via the SICB data cards file, the C++ algorithms via job options. This is discussed in Section 4.2.

3.2 Supported versions and compatibility with input data

Table 3.1 shows, for the currently supported versions of Brunel, the compatibility between Brunel versions and SICBMC / dbase versions used to produce the input data.

Please note that Brunel version v5r2 is supported only to allow backwards compatibility with the last major physics production ("Trigger TDR" production of summer 2001). All new studies should be performed using the latest Brunel version, v9r1. The features described in in this manual are those of version v9r1. Please refer to an older version of this manual to find the features of Brunel v5r2

Compatibility information for no longer supported versions of Brunel is shown in Table 3.2

Table 3.1 Version compatibility table for supported versions of Brunel

Brunel version	SICBMC version	dbase version	Remarks
v9r1	v249	v243r2, v243r3	"Realistic LHCb-light" with split, hybrid IT / OT TT1
		v243r1, v243r1p1	"Realistic LHCb-light" with split all Silicon TT1
		v242r1	"Realistic LHCb-light" with all Silicon TT1
		v242	"Realistic LHCb-light"
	v248*	v241	"LHCb-light"
	v247*	v240r1	"LHCb-minus" with tracking cross geometry, magnet stations removed
		v240	"LHCb-classic" with tracking cross geometry
v5r2	v246	v239	"LHCb-classic" geometry
	v245	v238	

Table 3.2 Version compatibility table for obsolete versions of Brunel

Brunel version	SICBMC version	dbase version
v9r0	v247* to v249*	v240 to v243r1
v8	v247*, v248*	v240, v240r1, v241
v7r1	v248	v241
v6r1	v247	v240
v3r1 to v5r1	v246	v239
	v245	v238
v3	v243 and v244	v237

3.3 Detector Description

Brunel takes detector description information both from the XML and SICB databases.

The XML database packages used by the currently supported versions of Brunel are: `DetDesc v9r2`, `XmlDDDB v8r1`, `CaloDet v3r0`, `VeloDet v3r1`.

The SICB database consists of the `SICB/detdes` package (`v14` for Brunel `v9r1`) and of the `SICB/dbase` package which also contains configuration information for the algorithms. The current version of Brunel can read data produced with several `dbase` versions, as shown in Table 3.1; the appropriate `dbase` version must be selected at run time, as described in Section 4.2.1.

3.4 Event Data Model

Event data can originate either from SICB banks in the ZEBRA common block (in particular, the RAWH data output by SICBMC), or as output from C++ reconstruction algorithms.

The conversion from SICB banks to Transient Event Data, for use by C++ Gaudi Algorithms, is performed by converters from the following packages: `SicbCnv v15r1`, `CaloSicbCnv v5r1`, `MuonSicbCnv v4r0`, `TrSicbCnv v4r0`, `VSicbCnv v6r1`.

The Transient Event Data Model is described in the following packages: `EventKernel v1r0`, `LHCbEvent v12r2`, `L0Event v10r0`, `L1Event v6r0`, `ITEvent v7r2`, `OTEvent v7r0`, `TrKernel v6r0`, `TrEvent v7r1`, `CaloKernel v1r0`, `CaloEvent v7r1`, `VeloEvent v6r0`.

3.4.1 PileUp

Since SICBMC v244, it is possible to add PileUp to events at the generator level. For this reason, the addition of PileUp in Brunel is not supported. We use the terminology that *PileUp* is due to multiple interactions in the current beam crossing . The effects on detector response of interactions occurring in preceding or subsequent beam crossings are called *SpillOver*.

3.4.2 SpillOver

Brunel can read more than one event into the Event Data Store. In addition to the main event (that may contain *PileUp*, as discussed in the previous section), additional *SpillOver* events may be read into parallel event data structures. By default, only the SpillOver data actually required by reconstruction algorithms is read in. These defaults are controlled by job options, as described in Section 4.2.2.1.

The spillover events are read into additional branches of the LHCb data model, as shown in Table 3.3. Note that the events are merely made available in the transient event data store, and

Table 3.3 SpillOver in the Transient Event Data Model

Beam crossing	Event path
Previous	/Event/Prev
One before previous	/Event/PrevPrev
Next	/Event/Next
One after next	/Event/NextNext

it is up to the digitisation algorithms to make use of this information if required. None of the event information is modified, in particular the time of flight information of the hits is not modified: the labels Prev, Next etc. are for convenience only, it is up to the algorithms using this infomation to add appropriate timing offsets when required.

3.4.2.1 Limitations

The current implementation of spillover has the following limitations:

- In order to determine the probability of interactions in previous and subsequent bunch crossings, the spillover algorithm takes the instantaneous luminosity from the current event (as used to generate PileUp). Based on this probability, it uses a random number generator to simulate the actual number of interactions in each of the bunch crossings (let's call this number `num_inter`). If, in a given bunch crossing, `num_inter` is greater than zero, then an event is read into the SpillOver transient event structure from the SpillOver input file. This approach is an approximation: if `num_inter` > 0, the SpillOver algorithm always reads one (and only one) event from the input file, regardless of the value of `num_inter`. In theory the event read from the input file should contain num_inter piled up events; in practice one just reads the next event. It would be possible to do the correct thing by skipping events until one with the right PileUp multiplicity is found, or to open several input files, each containing events with a fixed PileUp multiplicity. Neither of these possibilities is currently implemented.
- Since SpillOver events will be combined by digitisers into a single Raw event, it is only foreseen to provide the `/MC` part of the event. Furthermore, only those parts of the `/MC` subevent whose converter foresees SpillOver are available.

3.5 Random numbers

Random numbers are currently used in Brunel to smear MonteCarlo truth information ("cheated pattern recognition") and in the digitisation phase. In order to ensure reproducibility when reconstructing the same events in a different order, the random number seeds are re-initialised at every event.

Brunel uses two different random number engines. Fortran code (from SICB) uses the RANECU engine (see GRNDM routine in Futio package); C++ code uses the RanLux engine from CLHEP. The RANECU engine, which expects two seeds, is initialised with the run number and event number as seeds. The RanLux engine, which uses only one seed, is initialised with the run and event numbers combined according to the following formula:

```
long theSeed = 100000 * (evt->event() % 20000) + (evt->run() % 100000);
```

3.6 Digi Phase

The digitisation phase converts input MonteCarlo Hits data (RAWH, RAWH2) into digitised Raw data. The following sub-systems participate in this phase:

3.6.1 VELO

The Fortran digitisation from package `digvdet v2r3` (which uses an old Velo geometry) is executed in parallel with the C++ digitisation from package `VeloDigit v1r0` (which uses the Velo TDR geometry from the XML database). The output of the C++ algorithm can be accessed via the Transient Event Store path "`/Event/Raw/VeloClus`". The output of the Fortran algorithm (SICB banks VSCR, VSCP) can be accessed via the paths "`/Event/Rec/Velo/RClusters`", "`/Event/Rec/Velo/PhiClusters`".

Only the output of the Fortran algorithm is made persistent on the Brunel DST.

3.6.2 Inner Tracker

C++ digitisation from package `ITAlgorithms v8r1`. The output is only available in the Transient Data Store to C++ algorithms. The corresponding SICB banks WIDG are no longer available.

3.6.3 Outer Tracker

C++ digitisation from package `OTAlgorithms v7r0`. The output is only available in the Transient Data Store to C++ algorithms. The corresponding SICB banks WODG are no longer available.

3.6.4 RICH

Fortran digitisation from package `recrich v5r4` (`RIDIGI`).

3.6.5 CALOrimeters

C++ digitisation from package `CaloAlgs v4r3`. TDR geometry. Output is converted back to SICB banks ECEL, HCEL, ECPC.

3.6.6 MUON

Fortran digitisation from package `digmuon v4`. TDR geometry. The muon background can be simulated before running the digitisation (package `simmubg v5r3`). This can be switched on with the job option `BrunelDigMUON.addBkg = true;` (default is `false;`). Please note that, due to a bug in the code removing previously added background, it is not possible to add a different background to data which already had muon packground added in the SICBMC step.

3.7 Trigger phase

The Trigger phase simulates the execution of the L0 and L1 trigger algorithms in the DAQ. Therefore it is logically equivalent to the digitisation phase and is part of the simulation.

3.7.1 Technical Proposal (TP) trigger algorithms

Fortran L0, L1 and L2 trigger algorithms from the packages `trihadr v4, trilvl2 v5r1, trimuon v5, trit0v v5, triskel v4, trit1tr v3r3, trivert v5r3, trielec v5r1`

These packages implement the TP trigger algorithms, with the exception of the L1 track trigger, which is no longer available due to the absence of WIDG and WODG banks in the output of the tracking digitisation, and the L0 2x2 calorimeter trigger which is superseded by the C++ implementation.

3.7.2 L0 trigger

C++ L0 trigger simulation from the packages `L0Calo v4r0`, `L0Muon v4r0`, `PuVeto v1r0`, `L0DU v4r0`. The only output is the trigger decision, which is written into the PASS bank.

3.8 Reco Phase

3.8.1 Inner Tracker

C++ hit reconstruction from package `ITAlgorithms v8r0`. The output is NOT converted back to SICB banks

3.8.2 Outer Tracker

C++ hit reconstruction from package `OTAlgorithms v7r0`. The output is NOT converted back to SICB banks

3.8.3 Velo Tracking

C++ reconstruction of tracks in the Velo, starting from the output of the C++ digitisation. Package `VeloTrack v1r1`. The output is not converted back to SICB banks and therefore is not saved.

3.8.4 Forward tracking

C++ reconstruction of tracks in the spectrometer, starting from the Velo tracks and extrapolating them downstream, using the algorithm described in reference [10] (package `FwTrack v1r1`). The output is not converted back to SICB banks and therefore is not saved.

3.8.5 Upstream tracking and track fit

C++ track reconstruction from package `TrAlgorithms v7r1`. This consists of track seeding, track following, and track fit, with cheated pattern recognition (using MC truth information). The output is converted back to SICB banks AXAT, AXTP.

3.8.6 RICH

FORTRAN reconstruction, including "extended tracking", from package `recrich v5r4 (RIRECO)`.

3.8.7 CALOrimeters

FORTRAN reconstruction from packages `rececal v7 and rechcal v6r1`. Not tuned to latest geometry as used by the digitisation. The OO reconstruction algorithms from packages `CaloAlgs v4r3 and CaloCA v3r1` are also executed, but the output is not converted back to SICB banks and therefore is not saved.

3.9 Final Fit phase

This is where the last reconstruction pass is made, currently purely in FORTRAN from the `axreclib v4r3` package.

3.10 Moni Phase

The following monitoring algorithms are available. See Section 4.2.5 for details on how to control monitoring in Brunel.

3.10.1 ITMCHitsMonitor, ITDigitsMonitor

These C++ algorithms (from package `ITAlgorithms`) produce histograms to check the results of the Inner Tracker digitisation. The histograms are saved in the `ITMCHITSMONITOR`

and `ITGDIGITSMONITOR` sub-directories of the `TRACKING` directory of the histogram output file. Enabled by default

3.10.2 OTDigitChecker

This C++ algorithm (from package `OTAlgorithms`) produces histograms to check the results of the Outer Tracker digitisation. The histograms are saved in the `TRACKING/OTDIGITCHECKER` sub-directory of the histogram output file. Enabled by default

3.10.3 TrMonitor, TrCreat, TrFitIn

These C++ algorithms (from package `TrAlgorithms`) produce histograms to check the results of the track following and track fit. The histograms are saved in the `TRMONITOR`, `TRCREAT` and `TRFITIN` sub-directories of `TRACKING` directory of the histogram output file. Enabled by default.

3.10.4 L0CaloMonit

This C++ algorithm (from package `L0Calo`) produces histograms to check the results of the L0 calorimeter trigger simulation. The histograms are saved in the `L0` directory of the histogram output file. Enabled by default

3.10.5 ITDigitChecker

This C++ algorithm (from package `ITAlgorithms`) produces an ntuple to check the results of the Inner Tracker digitisation. Since it is undesirable to produce ntuples in a production environment, this algorithm is not enabled by default.

3.10.6 FwtAnalyse

This C++ algorithm (from package `FwTrack`) produces an ntuple to check the results of the forward tracking algorithm. Since it is undesirable to produce ntuples in a production environment, this algorithm is not enabled by default

3.10.7 SpdMonit, PrsMonit, EcalMonit, HcalMonit

These C++ algorithm (instances of `CaloDigitMonitor` from package `CaloAlgs`) produce histograms to check the results of the calorimeter digitisation. The histograms are saved in the `SPD, PRS, ECAL, HCAL` directories of the histogram output file. Enabled by default

3.11 Output data

The current output data format of Brunel is in the form of DST (or DST2 depending on the input data) ZEBRA files, containing the standard SICB DST information. None of the transient event data is saved as objects: only data converted back to SICB banks is saved.

To save space, certain SICB banks are either dropped or compressed prior to saving the DST. This is done by the Fortran routine `dropbanks.F`, reproduced below.

Listing 3.1 Banks dropped or compressed before saving the DST

```
      call ubdrop('E2RW')
      call ubdrop('E3RW')
      call ubdrop('E4RW')
      call ubdrop('H1RW')
      call ubdrop('RIDT')
      call ubpress('ECMT','R')
      call ubpress('HCMT','R')
```

The dropping of banks can be disabled, as explained in Section 4.2.3.

Chapter 4
Customising and Running Brunel

4.1 Introduction

The released versions of Brunel contain the recommended configuration for a standard LHCb reconstruction job. This chapter describes how to execute a standard job, how to modify the run time behaviour of Brunel, and how to modify its functionality by adding or removing code.

4.2 Modifying the run time behaviour

Even if you wish to run a standard job, you will need to make some modifications, if only to define the input and output event data files of your reconstruction job.

The `/options` subdirectory of the Brunel package contains several files with a name like `Brunel<dbver>.opts`, where `<dbver>` is a SICB database version number (e.g. `Brunelv243r1.opts`). There is one such file for each SICB database supported by the current version of Brunel. This is the top level file that should be passed to the job, as discussed in Section 4.4. Users should modify the file corresponding to the database they wish to use (i.e. consistent with the input dataset) to e.g. define the input data, as discussed in the next subsections.

Every one of these database dependent top level files include the file `Brunel.opts` which contains the default Brunel configuration common to all supported database versions, and which should not need to be modified by most users. This file in turn includes a number of other files that could be modified to alter the default behaviour. These files are listed in Table 4.1 and their contents are also described in the next subsections.

The `/options` subdirectory also contains a `Brunel.cards` file. This file is a SICB data cards file that can be used to modify the behaviour of the SICB algorithms exceuted within Brunel. Any SICB data card may be used, with the exception of cards dealing with input event data

Table 4.1 Job options files included by the main `Brunel.opts` job options file

Job Options File	Purpose
BrunelMessage.opts	Modify the printing behaviour
BrunelMoni.opts	Enable / Disable monitoring

(TRIGGERS card, IOPA 'GETX', 'GETY', 'GETZ' cards) and selection of processing steps (SKIP data card), since this functionality is handled by Gaudi. Please refer to the SICB documentation [8] for details of available cards.

4.2.1 Database selection

The Brunel requirements file does not depend explicitly on a specific version of the SICB geometry and configuration database (`SICB/dbase`), because Brunel is able to process data produced by several combinations of SICBMC and `dbase` versions. In order to configure Brunel correctly at run time, the environment variable `$LHCBDBASE` must be explicitly set to point to the `dbase` version used to produce the Brunel input data. This can be done from the command prompt, for example for `dbase v241`:

```
setenv LHCBDBASE $LHCBSOFT/SICB/dbase/v241         // Linux

set LHCBDBASE=%LHCBHOME%\software\NEW\SICB\dbase\v241   // Windows DOS prompt
```

The Brunel main program uses the `$LHCBDBASE` environment variable to determine the name of the `Brunel<dbver>.opts` file it should open in order to get its job options. This default can of course be over-ridden, as described in Section 4.4.2.

Note that on Windows, Developer Studio must be started from the DOS prompt where `LHCBDBASE` was defined (simply type `msdev` at the prompt). On Linux, the script `Brunel.job` can be used, giving the database version as the first argument - e.g.:

```
bsub -q z5_8nh Brunel.job v241
```

4.2.2 Defining input data

Brunel uses the Gaudi `EventSelector` to read in event data from a SICBMC RAWH (or RAWH2 or RAWH3) file. The relevant job options should be set in the top level database dependent `Brunel<dbver>.opts` file by changing the lines that are given as example in Listing 4.1

Listing 4.1 Job Options for event input definition

```
1:  EventSelector.Input = {"JOBID='52310'"}; // Input file name
2:  // Number of events to be processed (default is all events)
3:  EventSelector.EvtMax = 100;
4:  // EventSelector.FirstEvent = 3; //Uncomment to skip some events
```


4.2.2.1 Enabling SpillOver

The `Brunel<dbver>.opts` file also contains examples of the options needed to switch on the reading of SpillOver events into Brunel. These lines are reproduced in Listing 4.2.

Listing 4.2 Job Options for spillover

```
1:  //#include "$SICBCNVROOT/options/Brunel.opts" //uncomment for Spillover
2:  //SpillOverSelector.Input = {"JOBID='86160'"}; //evtype 61 for Spillover
```

By default, spillover is disabled. To enable it, uncomment Line 1: this reads in the default spillover configuration for Brunel from the `SicbCnv` package. You also need to provide an input file of minimum bias events (RAWH or RAWH2, produced with the correct `dbase` version) by uncommenting and modifying Line 2.

The default spillover configuration for Brunel is shown in Listing 4.3.

Listing 4.3 Default settings for SpillOver from the file `$SICBCNVROOT/options/Brunel.opts`

```
 1:  SpillOverAlg.SpillOverPrev = 2;
 2:  SpillOverAlg.SpillOverNext = 1;
 3:  //----------------------------
 4:  // Data to be loaded
 5:  SpillOverAlg.SpillOverData = {
 6:    "MCOuterTrackerHits", "MCInnerTrackerHits",
 7:    "Prs/Signals",  "Prs/SummedSignals",
 8:    "Spd/Signals",  "Spd/SummedSignals",
 9:    "Ecal/Signals", "Ecal/SummedSignals",
10:    "Hcal/Signals", "Hcal/SummedSignals"  };
```

The first two lines indicate that the `SpillOver` algorithm will read in events for two beam crossings preceding and one beam crossing following the current event. Lines 5 to 10 specify the data to be loaded. Only data actually expected by Brunel algorithms needs to be requested. The current default reflects the requirements of the algorithms in the current Brunel version. To modify these defaults you should modify the file `$SICBCNVROOT/options/Brunel.opts` and include the modified file in `Brunel<dbver>.opts`. Note that only data whose SICB converter has foreseen SpillOver can be added in this way.

4.2.3 Defining output data

The Gaudi framework does not provide a facility for writing out event data to ZEBRA files. For this reason, Brunel calls the SICB routine RECEVOUT to write out the SICB DST file. The output stream is defined in the `Brunel.cards` file using an IOPA 'SAVX' data card as for SICB (see Listing 4.4).:

Listing 4.4 SICB card to define the ZEBRA output file

```
1:  IOPA
2:      'SAVX' 'XO' '$WORKDIR/Brunel.dst!'
```

By default, a certain number of SICB banks are dropped before writing out the DST. The current list of dropped banks is given in Listing 3.1. It is possible to disable the dropping of

these banks, by setting the job option `BrunelFinish.DropDSTBanks` to `false` in the file `BrunelInput.opts`.

```
// Uncomment next line to keep all SICB banks on DST output
//BrunelFinish.DropDSTBanks = false;
```

It is also possible to write out an object-oriented DST to a ROOT file, using the facilities provided by Gaudi. Please refer to the Gaudi manual [1] for details.

4.2.4 Modifying the printing behaviour

Brunel uses the Gaudi Message Service to print out information. The amount of information to be printed is controlled by job options. The file `BrunelMessage.opts` sets up the default printing behaviour - you should modify this file if you wish to change the defaults. An extract of this file is shown in Listing 4.5.

Listing 4.5 The `$BRUNELROOT/options/BrunelMessage.opts` job options file

```
 1: // Print event number at every event
 2: EventSelector.PrintFreq = 1;
 3:
 4: // Modify Message Format to print algorithm name with 80 characters
 5: //MessageSvc.Format = "% F%80W%S%7W%R%T %0W%M";
 6:
 7: //-------------------------------------------------------------
 8: // Output thresholds (2=DEBUG, 3=INFO, 4=WARNING, 5=ERROR, 6=FATAL )
 9: //-------------------------------------------------------------
10: MessageSvc.OutputLevel = 4;
11:
12: // Print Chrono statistics at end of job
13: ChronoStatSvc.OutputLevel = 3;
14: ChronoStatSvc.ChronoPrintLevel = 4;
15: ChronoStatSvc.StatPrintLevel = 4;
16:
17: // Suppress excessive warnings from some algorithms
18: TrFit.OutputLevel = 5;
19: TrMonitor.OutputLevel = 5;
```

Line 2 controls the frequency at which the event number is printed out at the beginning of the event loop: the default is every event. Line 5 can be used to modify the format of the messages printed out by the message service. Compared to the default behaviour of Gaudi, this line prints the name of the algorithm generating the message in a field 80 characters wide. This can be done to avoid truncating the name of the many Brunel algorithms with long names; on an 80 column screen the name will appear on one line and the message on the following line(s). Line 10 sets the default print level to WARNING: only messages flagged WARNING or above will be printed. Sometimes it may be desirable to over-ride this default for certain algorithms or services - the current Brunel setting is shown in Lines 13 to 19.

Control of debug printout from the SICB Fortran algorithms used in Brunel is via DEBU, PRNT and DEBG data cards to be added to the file `Brunel.cards`.

4.2.4.1 Rules for printout from Brunel algorithms

Developers of algorithms to be used in Brunel should remember that their code will be used in production, and therefore that printout should be kept to a bare minimum. The following rules are suggested:

- All printout must be via the Gaudi Message Service: use of std::cout is forbidden.
- The algorithm name used to create the the `MsgStream` object should be kept short - this name is printed out to the log file and by default is truncated to 18 characters.
- Any messages introduced to debug the code should use the DEBUG output level
- INFO level messages should not be used in the execute() method of algorithms (i.e. in the event loop). A possible exception would be to flag errors which are not fatal to the algorithm's execution (e.g. if the track fit fails on a particular track). It should not be used to flag "normal" errors (e.g. an empty hits container).
- WARNING, ERROR and FATAL messages are reserved for real problems that cause an algorithm to return an error. An explanatory message must always be printed whenever and algorithm does not behave as expected. It is suggested to use WARNING when the problem affects only the current algorithm, ERROR if it affects the current event (i.e. that processing of the current event should stop), and FATAL if it affects the rest of the job (i.e. that the job should stop).

4.2.5 Monitoring options

A number of possibilities exist to monitor the execution of Brunel. These are steered by job options in the file `BrunelMoni.opts` and by SICB data cards in the file Brunel.cards.

4.2.5.1 Monitoring histograms filled by C++ code

Brunel algorithms book and fill histograms via the standard Gaudi histogram service. These histograms can be saved either as ROOT or HBOOK histograms, as described in the Gaudi Users Guide. By default, Brunel saves these histograms in HBOOK format, in a file called `Histos.hbook`, as shown in Listing 4.6. Remove lines 2 and 3 if you wish to suppress the printing of histograms, or replace them with the equivalent lines for ROOT if you wish to save them in ROOT format.

Listing 4.6 Histogram persistency options from the file `BrunelMoni.opts`

```
1: // Hbook persistency (use HBookCnv v* in requirements)
2: #include "$STDOPTS/Hbook.opts"
3: HistogramPersistencySvc.OutputFile = "Histos.hbook";
```

Of course, suppressing the printout of histograms does not prevent their filling. The time spent filling the histograms would then be wasted. In a program used in production, it would be preferable to be able to suppress the filling of histograms. This has been foreseen in Brunel by instantiating a Brunel phase, `BrunelMoni`, dedicated to the filling of monitoring histograms. Monitoring histograms should be filled, wherever possible, by dedicated monitoring algorithms executed in this `BrunelMoni` phase. It is then simple to suppress the monitoring histograms by not executing this phase in a production job. By default, Brunel

instantiates the `BrunelMoni` phase and executes it as the last phase of the event loop. To suppress monitoring histograms, remove line 1 of Listing 4.7 from the file `BrunelMoni.opts`..

Listing 4.7 Monitoring options from the file BrunelMoni.opts

```
 1:  ApplicationMgr.TopAlg += { "BrunelPhase/BrunelMoni" };
 2:  //-------------------------------------------------------------
 3:  // Detectors to monitor
 4:  //-------------------------------------------------------------
 5:  BrunelMoni.DetectorList = { "IT", "OT", "Tr" };
 6:  //-------------------------------------------------------------
 7:  // Monitoring algorithms
 8:  //-------------------------------------------------------------
 9:  BrunelMoniOTSeq.Members   = { "OTDigitChecker" };
10:  BrunelMoniTrSeq.Members   = { "TrMonitor" };
```

The remaining lines of Listing 4.7 set up the sequences and algorithms of the `BrunelMoni` phase, as described in Sections 2.3 and 3.10.

4.2.5.2 Monitoring histograms filled by SICB subdetector code

Any of the histograms filled by the SICB reconstruction code can be filled and saved in the standard way, by providing the appropriate SICB data cards in the file `Brunel.cards`, for example:

```
IOPA
    'RHIS' 'HO' 'rich.hbook!'
    'T1VE' 'HO' 'trigL1.hbook!'
```

4.2.5.3 Profiling

Brunel makes use of Gaudi Auditors to monitor the code performance at run time. The following auditors are available:

NameAuditor Prints out the name of an algorithm whenever its `execute()` method is called. Disabled by default.

ChronoAuditor Monitors CPU usage of each algorithm and reports at the end of the job the total and average time per algorithm. Enabled by default.

MemoryAuditor Prints out information on memory usage, in particular whenever the memory allocation changes. Currently only works on Linux. Disabled by default.

The default behaviour of these auditors can be changed using the following job options in the file `BrunelMoni.opts`:

```
AuditorSvc.Auditors  = { "NameAuditor", "ChronoAuditor", "MemoryAuditor" };
NameAuditor.Enable   = false;
ChronoAuditor.Enable = true;
MemoryAuditor.Enable = false;
```


4.2.6 Enabling static execution

If you wish to execute the statically linked version of Brunel, you need to over-ride all the job options that define the DLLs to be loaded by the application manager. This is done by uncommenting the following line in the file `Brunel<dbver>.opts`:

```
//ApplicationMgr.DLLs = {"NONE"};
```

4.2.7 Additional job options

An additional user job options should be defined near the end of the file `Brunel<dbver>.opts`. Since the parsing of job options is sequential, any of the options previously defined can be redefined here.

4.2.7.1 Suppressing reconstruction phases

You may wish to suppress one of the reconstruction phases by redefining the `ApplicationMgr.TopAlg` option shown in Listing 2.1.

A special case is if you wish to use Brunel simply as an analysis framework, switching off all the reconstruction phases. In this case, the `ApplicationMgr.TopAlg` option would become:

```
1:  ApplicationMgr.TopAlg = { "BrunelInitialisation/BrunelInit",
2:                            "MyAlg",
3:                            "BrunelFinalisation/BrunelFinish" };
```

Line 1 is necessary to correctly initialise the event in the ZEBRA memory; line 2 adds a private C++ analysis algorithm `MyAlg`, while line 3 is necessary to invoke the Fortran user analysis routine SUANAL. Of course it is not necessary to provide both lines 2 and 3, it depends on the application.

4.2.7.2 Controlling the muon background

By default, Brunel removes any previously added muon background from the input RAWH file, and regenerates the muon background according to the parameterisation defined by the `MUBG` and `MUBC` cards in the standard.stream file of the SICB database. This is done in the digitisation phase, before creating the muon digitisings. To suppress the muon background simulation, you should add the following job option to `BrunelUser.opts`:

```
BrunelDigiMUON.addBkg = false;
```


4.3 Adding user code

User code should be added to an existing Brunel Phase. The way to do this depends on the packaging of the algorithm to be added:

1. If the new algorithm is part of a package already known to Brunel, it is sufficient to add the algorithm to the appropriate sequence, in the package specific `Brunel.opts` file (see for example Listing 2.3).
2. If the new algorithm is part of a package not yet known to Brunel, the new package should provide a `Brunel.opts` file in the `/options` subdirectory. This file should have a structure similar to that in Listing 2.3 and be included in the main `Brunel.opts` file (or near the end of the `Brunel<dbver>.opts` file), as shown in Listing 2.4. You should of course `use` the new package in the Brunel CMT `requirements` file.
3. It is also possible to add a Fortran analysis routine, using the SICB user routines SUINIT, SUANAL, SULAST. SUANAL is called at the end of all event processing. These routines should be linked into the application as shown for example in Listing 4.8.

Listing 4.8 Example or `requirements` for linking user FORTRAN code into Brunel

```
1:  application Brunel ../src/BrunelMain.cpp \
2:                     $(BRUNELSICBROOT)/src/Objs/*.F \
3:                     ../src/MyAnal/*.F
```

In this example the user analysis code (including the routine SUANAL) has been saved in the `MyAnal` subdirectory of the Brunel package.

4.4 Building and running the job

This section gives simple instructions on how to execute a Brunel job. Familiarity is assumed with CMT [9]. The instructions are given for Linux at CERN. Windows procedures are similar, the difference should be fairly obvious to anyone who is familiar with the Windows development environment. An example job for executing Brunel on Linux in both interactive and batch environments is distributed with Brunel in the `/job` subdirectory. You should tailor it to your needs, using the information below.

Note that the LHCBDBASE environment variable must always be set before running a Brunel job. Thiis was described in Section 4.2.1.

4.4.1 Running the default version

If you wish to execute the default version of Brunel, without changing any of the job options, you simply have to set up all the necessary environment variables and then execute the job:

```
cd ~/myBrunelTest
source $LHCBSOFT/Rec/Brunel/v9r1/cmt/setup.csh
setenv LHCBDBASE $LHCBSOFT/SICB/dbase/v241
$BRUNELROOT/rh61_gcc2952/Brunel.exe > myjob.log
```

This is useful to check that your environment is set correctly. The file `myjob.log` should be identical to the sample output in the production area: `$BRUNELROOT/job/linux.log` (except of course for differences due to the execution time of the two jobs). The job should also produce an hbook histogram file in the current directory, whose contents should be identical to the sample histogram output in the production area: `$BRUNELROOT/job/linux.hbook` (similar sample files `win.log` and `win.hbook` exist for the Windows platform)

4.4.2 Running the default version with modified job options

In real life you will certainly need to modify the job options, if only to change the name of the input file. In this case you should copy the job options directory from the official area, edit one or more files, and change the logical name pointing to these files:

```
cd ~/myBrunelTest
source $LHCBNEW/Rec/Brunel/v9r1/cmt/setup.csh
cp $BRUNELROOT/options/*.* .
emacs ...
setenv BRUNELOPTS .
setenv SICBCARDS ./Brunel.cards
setenv LHCBDBASE $LHCBSOFT/SICB/dbase/v241
$BRUNELROOT/rh61_gcc2952/Brunel.exe > myjob.log
```

If you have changed the name of the top level job options file (useful if you need to launch several jobs in parallel), you also need to tell Brunel where to find the modified file, either before executing the job:

```
setenv JOBOPTPATH ./myBrunel.opts
```

or as an argument to the job:

```
$BRUNELROOT/rh61_gcc2952/Brunel.exe ./myBrunel.opts > myjob.log
```

Note that these instructions are valid also if you want to change the Brunel functionality by excuting only a subset of the standard algorithms, sequences or phases: it is sufficient to make the necessary changes to the job options. Similarly if you want to add an algorithm from a component library already known to Brunel.

4.4.3 Running the default version with modified requirements

If you wish to use a new version of an existing component library, or use algorithms from a component library not yet known to Brunel, you will need to modify the running environment of Brunel. In this case it may be sufficient to modify the CMT requirements file and rebuild the Brunel environment, without actually rebuilding the Brunel executable::

```
cd ~/newmycmt
getpack Rec/Brunel v9r1
cd Rec/Brunel/v9r1/cmt
emacs requirements
...
source setup.csh
setenv LHCBDBASE $LHCBSOFT/SICB/dbase/v241
cd ../job
$LHCBNEW/Rec/Brunel/v9r1/rh61_gcc2952/Brunel.exe > myjob.log
```

4.4.4 Building a modified version

In most cases, developers will need to build a new Brunel executable. Since Brunel is a standard CMT package, it is sufficient to type `gmake` in the `/cmt` sub-directory. The procedure becomes:

```
cd ~/newmycmt
getpack Rec/Brunel v9r1
cd Rec/Brunel/v9r1/cmt
emacs requirements
...
source setup.csh
setenv LHCBDBASE $LHCBSOFT/SICB/dbase/v241
gmake
cd ../job
../rh61_gcc2952/Brunel.exe > myjob.log
```

4.5 Problem reporting and resolution

4.5.1 Known Problems

The following problems and workarounds are known:

- On Windows, the `LHCBHOME` environment variable must contain a path with at least one backslash (e.g. "`%SITEROOT%\lhcb`"). If not, ZEBRA will complain when trying to open the file `$LHCBHOME/sim/data/v111-prob-2d-d0.hbook`. This is a feature of the shift library for Windows..

- With the static executable of Brunel, it is not possible to save histograms produced by C++ algorithms via the Gaudi histogram service.

4.5.2 Getting help

If your problem cannot be resolved by looking at this guide, you could try using the LHCb software discussion mailing list, *lhcb-soft-talk@cern.ch*.

4.5.3 Reporting problems

If you think you have found a bug in Brunel or in Gaudi, or if you would like to request a new feature, please use the LHCb problem reporting system:
http://cern.ch/hep-service-prms/lhcb.html

Appendix A
References

1 The GAUDI users guide is available at:
http://cern.ch/lhcb-comp/Frameworks/Gaudi/Gaudi_v9/GUG/Output/GUG.htm

2 CMT documentation is available at *http://cern.ch/lhcb-comp/Support/html/cmt.htm*

3 A compendium of LHCb notes concerning reconstruction is available at:
http://cern.ch/lhcb-comp/Reconstruction/LHCbNotesOfInterest.html

4 Information about the LHCb Event Data model is available at:
http://cern.ch/lhcb-comp/Frameworks/EventModel/

5 See for example *http://www.gaudiclub.com/ingles/i_vida/i_menu.html* for more information about Antoni Gaudi

6 See for example *http://www.spartacus.schoolnet.co.uk/RAbrunel.htm* for more information about Isambard Kingdom Brunel

7 Sub-detector job options for Brunel,
http://cern.ch/lhcb-comp/Support/Conventions/options.pdf

8 The SICB documentation is available at: *http://cern.ch/lhcb-comp/SICB/*

9 Documentation on CMT and on its use within LHCb is available at:
http://cern.ch/lhcb-comp/Support/html/cmt.htm

10 The forward tracking, an optical model method. LHCb note 2002-008.
http://weblib.cern.ch/abstract?LHCb-2002-008@LHBLHB