BRUNEL

LHCb Reconstruction Program

User Guide

Corresponding to Brunel version v3r1

Version: 3.1
Issue: 0
Date: 5 June 2001

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:	3.1		
	Issue:	0		
	Edition:	0		
	ID:	[Document ID]		
	Status:			
	Created:	25 May 2000		
	Date:	5 June 2001		
	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**
1	0	26 May 2000	First draft version
1	1	28 July 2000	Minor changes for Brunel v1r2
1.5	1	17 Nov. 2000	Updated for Brunel v1r5, GaudiSys v6
1.6	1	8 Jan 2001	Updated for Brunel v1r6
3	0	20 May 2001	Updated for Brunel v3
3	1	28 May 2001	Fix a few typos and documentation errors
3.1	0	5 June 2001	Updated for Brunel v3r1

Table of Contents

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 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].

1.2 Package structure

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

- `src` C++ and Fortran source code
- `doc` release notes
- `job` example job
- `options` default job options and SICB data cards
- `cmt` CMT `requirements` file
- `Visual` Visual Studio Workspace

The `src` directory contains two sub-directories:

- `BrunelSicb` Brunel code and inteface code to SICB
- `ObsoleteSicb` Obsolete interface code to SICB (replaced by new C++ implementation)

Please note that the packaging was modified starting from Brunel v3. A top level (i.e. not under the `Rec` hat) Brunel package exists in CVS, but this is obsolete and is no longer maintained.

1.3 What does "Brunel" mean?

All LHCb data processing applications are based on a framework which enforces the GAUDI architecture. Antoni Gaudi [4] 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 [5] 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 v3. 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. This Gaudi algorithm is where all initializations which are independent of BrunelPhase are performed. These can be global program initializations (in the `initialize()` method), or event by event initializations (in the `execute()` method). Note that initializations specific to a given BrunelPhase should not be performed here.

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.

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.

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

2.3 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.

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 2 to 5, 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 = { "BrunelInitialisation/BrunelInit",
2:                            "BrunelPhase/BrunelDigi",
3:                            "BrunelPhase/BrunelTrigger",
4:                            "BrunelPhase/BrunelReco",
5:                            "BrunelPhase/BrunelFinalFit",
6:                            "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 the current version 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" , "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 [6] 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",
                           "TrFitInitializer",
                           "TrEventTracksFitter"};

#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 current version of Brunel is a "hybrid" version, 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 Current functionality

3.2.1 Compatibility with input data

Table 3.1 shows the compatibility between Brunel versions and SICBMC / dbase versions used to produce the input data

Table 3.1 Version compatibility table

Brunel version	SICBMC version	dbase version
v3r1	v245 or later	v238 or later
v3	v243 and v244	v237

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*.

3.2.2 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.

The spillover events are read into additional branches of the LHCb data model, as shown in Table 3.2. Note that the events are merely made available in the transient event data store, and

Table 3.2 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.

3.2.2.1 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.

3.2.3 Digi Phase

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

3.2.3.1 VELO

Fortran digitisation from package `digvdet v2r2`. TDR geometry.

3.2.3.2 Inner Tracker

C++ digitisation from package `ITAlgorithms v4`. The output is NOT converted back to SICB banks WIDG, which are no longer available.

3.2.3.3 Outer Tracker

C++ digitisation from package `OTAlgorithms v4`. The output is NOT converted back to SICB banks WODG, which are no longer available.

3.2.3.4 RICH

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

3.2.3.5 CALOrimeters

C++ digitisation from package `CaloAlgs v2`. TDR geometry. Output is converted back to SICB banks ECEL, HCEL, ECPC.

3.2.3.6 MUON

Fortran digitisation from package `digmuon v4`. TDR geometry.

3.2.4 Trigger phase

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

These packges 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.

3.2.5 Reco Phase

3.2.5.1 Inner Tracker

C++ hit reconstruction from package `ITAlgorithms v4`. The output is NOT converted back to SICB banks

3.2.5.2 Outer Tracker

C++ hit reconstruction from package `OTAlgorithms v4`. The output is NOT converted back to SICB banks

3.2.5.3 Tracking system

C++ track reconstruction from package `TrAlgorithms v4r1`. 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.

3.2.5.4 RICH

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

3.2.5.5 CALOrimeters

FORTRAN reconstruction from packages `rececal v7 and rechcal v6`. Not tuned to latest geometry as used by the digitisation.

3.2.6 Final Fit phase

This is where the last reconstruction pass is made, currently purely in FORTRAN from the `axreclib v4r1` package.

3.2.7 Moni Phase

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

3.2.7.1 OTDigitChecker

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

3.2.7.2 TrMonitor

This C++ algorithm (from package `TrAlgorithms`) produces histograms to check the results of the track following and track fit. The histograms are saved in the `TRACKING` directory of the Histogram Service output file. Enabled by default.

3.2.7.3 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.

3.2.8 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('E1RW')
      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 version of Brunel contains 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 a main `Brunel.opts` file that should not need to be modified by most users. This file includes a number of other files that should be modified to alter the default behaviour. These files are listed in Table 4.1 and their contents are described in the next subsections.

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

Job Options File	Purpose
BrunelInputs.opts	Define input data and SpillOver mode
BrunelMessage.opts	Modify the printing behaviour
BrunelMoni.opts	Enable / Disable monitoring
BrunelStatic.opts	Enable / Disable static execution mode
BrunelUser.opts	Hook for additional user defined job options

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 (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 [7] for details of available cards.

4.2.1 Defining input data

The current version of Brunel uses the Gaudi EventSelector to read in event data from a SICBMC RAWH (or RAWH2) file. The relevant job options are found in the BrunelInput.opts file and are reproduced in Listing 4.1

Listing 4.1 Job Options for event input definition

```
1:  // Input file name (all on one line!)
2:  EventSelector.Input = {"JOBID='44814'"};
3:  // Number of events to be processed (default is all events)
4:  EventSelector.EvtMax = 100;
5:
6:  // Enable next card if you wish to skip some events
7:  // EventSelector.FirstEvent = 3;
```

Note that Brunel version v3 expects input data produced with SICBMC v243 or greater.

4.2.2 Enabling SpillOver

The BrunelInput.opts file also contains 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:  // SPILLOVER: Uncomment next two lines to add spillover events.
2:  //-------------------------------------------------------------
3:  // #include "$SICBCNVROOT/options/Brunel.opts"
4:  // SpillOverSelector.Input = {"FILE='SICBMC_v244_mbias_1.rawh'"};
```

By default, spillover is disabled. To enable it, uncomment Line 3: 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 SICBMC v243 or greater) by uncommenting and modifying Line 4.

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

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

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"  };
```

`$SICBCNVROOT/options/Brunel.opts` and include the modified file in `BrunelInput.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.

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

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 = 3;
11: ToolSvc.OutputLevel    = 3;
12: OTDigitize.OutputLevel = 3;
```

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 INFOrmational messages: all messages flagged INFO or above will be printed. Lines 11 and 12 are temporary, they are needed to work around a bug in the current version of Gaudi

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`. The remaining lines of Listing 4.7 set up the sequences and algorithms of the `BrunelMoni` phase, as described in Sections 2.3 and 3.2.7.

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" };
```


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
C T1VE are SICBDST L1 trigger  monitoring histograms
    'T1VE' 'HO' '$LHCBHOME/scratch/Brunel/T1VE_brunel.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 modifying the file BrunelStatic.opts, by uncommenting the line:

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

4.2.7 Additional user job options

An additional dummy job options file, `BrunelUser.opts`, is provided to allow users an additional hook for modifying the job options without having to touch the main job options file `Brunel.opts`. Since this file is the last file to be included, any of the options previously defined can be redefined here. For example 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.

4.3 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 in the `BrunelUser.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 $(GAUDICONFROOT)/src/GaudiMain.cpp \
2:                     ../src/BrunelSicb/*.F ../src/BrunelSicb/*.cpp
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. 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 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.

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 $LHCBNEW/Rec/Brunel/v3/cmt/setup.csh
$BRUNELROOT/i386_linux22/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/v3/cmt/setup.csh
cp $BRUNELROOT/options/*.* .
emacs ...
setenv BRUNELOPTS .
setenv SICBCARDS ./Brunel.cards
$BRUNELROOT/i386_linux22/Brunel.exe > myjob.log
```

If you have modified the main job options file `Brunel.opts` (not recommended), you also need to tell Brunel where to find the modified file, before executing the job:

```
setenv JOBOPTPATH ./Brunel.opts
```


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 v3
cd Rec/Brunel/v3/cmt
emacs requirements
...
source setup.csh
cd ../job
$LHCBNEW/Rec/Brunel/v3/i386_linux22/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 v3
cd Rec/Brunel/v3/cmt
emacs requirements
...
source setup.csh
gmake
cd ../job
../i386_linux22/Brunel.exe > myjob.log
```

4.5 Known problems

The following problems and workarounds are known:

- When building Brunel on Linux, you have to `source setup.csh` before typing `gmake`. If you do not do this, `gmake` will not find the source file of the main program.

- 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.

Appendix A
References

1 The GAUDI users guide is available at: *http://cern.ch/lhcb-comp/Frameworks/Gaudi/Gaudi_v7/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 See for example *http://www.gaudiclub.com/ingles/i_vida/i_menu.html* for more information about Antoni Gaudi

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

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

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