Tom Junk
December 2, 1995

The Compressed MiniDST

What It Is

The "compressor" is really just three routines that pack, unpack, and optionally convert between machine formats, data from a pre-specified set of banks "the MiniDST," for input/output to a Jazelle data file. These routines are currently installed in DUCSDST, and are called

PHMKBM.PREPMORT -- the "compressor"
PHLKBM.PREPMORT -- the "decompressor"
PHCVBM.PREPMORT -- the "converter."

In addition, there is an associated template, PHMTOC.TEMPLATE, which defines a "table of contents" for the MiniDST. It is in this structure that a listing of how many of each kind of bank is present in the compressed record. Because the structure of the compressed MiniDST has evolved to meet user requests, and because the structure of the table of contents could not change while maintaining backward compatibility, this "table of contents" is an incomplete list of everything in the compressed data list. The following banks are in the MiniDST standard and currently are "compressed:"

MCHEAD
MCPART
MCPNT
MCPQRK
MCBEAM
PHEVCL
PHBM
PHPSUM
PHPOINT
PHCHRG
PHCRID
PHKLUS
PHKTRK
PHKCHRG
PHKELID
PHKMC1
PHWIC
PHWMC

An additional bank, PHKHIT, is currently under consideration for addition to the MiniDST. The MiniDST standard was developed by a committee (Mike Strauss, Homer Neal, Richard Dubois, Tony Johnson, Tracy Usher, Tom Junk, and others) with input from intended users of the SLD data. Its purpose is to provide a set of data of maximum utility to a maxiumum number of analyses while fitting on the SLD VMS cluster's disks. After the MiniDST was devised, it soon was discovered that it took 40 Kb per event to store the Monte Carlo, and required 100 ms per event to read in on a VAX 4000. The compressor was written in order to pack the data in the banks more tightly and to avoid Jazelle I/O overhead when reading in the data and creating banks. The tradeoff was to sacrifice flexibility in order to gain speed and space. The current performance of the compressor is to pack MC events into 50% of the space, data events into 75% of the space, and to speed read-in by a factor of ~5. The price is that the procedure depends heavily

on features of the data structure, and changes to the data structures require adjustments to the compressor, decompressor, and converter code, while being careful to maintain backward compatibility.

The compressor, decompressor, and converter all are currently installed in DUCSDST, although they are linked into SLDSHR. The reason for this is to allow IDA users to unpack the MiniDST records without loading DSTSHR and all of the shareables underneath it (including most subsystem shareables). See DUCSSLD:SLDSHR.OPT for details. Another routine, PHMDST.PREPMORT is necessary to register the addresses of the three compressor routines using JZIART. Jazelle needs to call the compressor routines, but since user shareables are linked above the Jazelle shareable, the routines addresses must be registered with JZIART. PHMDST is called at IDA startup.

To use the compressor from IDA, one simply requests the reserved seglist item MINIDST:

```
SEGLIST CREATE OUTLIST MINIDST
```

This way, each write statement will put one compressed MiniDST record to the outfile. No special action is required to read in a file of compressed MiniDST -- the format is automatically detected by Jazelle and sent to the unpacker. All JAZZDATA files contain a flag to indicate whether they were created on a VAX or on SLACVM. If the file is being read in on a machine different from what it was written on, the conversion routine is called.

To write out special versions of the MiniDST, say, with additional uncompressed banks, simply add to the output seglist:

```
JLIST CREATE MYJLIST MYBANK1 MYBANK2
SEGLIST CREATE OUTLIST MINIDST @MYJLIST
```

will cause write statements to write out records containing compressed MiniDST banks in addition to the extra banks. A word of caution is in order here. If the additional banks MYBANK1 and MYBANK2 have pointers to banks in the compressed MiniDST record, these will not be correctly filled in when the banks are read back in (MYBANK1 and MYBANK2 may contain pointers themselves and each other, though). This happens because relative addresses are not preserved when banks are compressed and decompressed. The decompressed banks may appear anywhere in memory upon readin, and it is part of the decompressor's job to rewire all of the pointers within the MiniDST list to point properly, but it cannot anticipate every user's bank. If such information needs to be kept in user banks, it is best to write bank ID's instead of pointers.

How It Works

The idea is to store all of the information in the banks listed above into a single contiguous block of memory in as compact a manner as possible, and also one for which minimum effort is necessary to reconstruct the original set of banks. Relative locations in memory need not be preserved, but all pointers within the MiniDST (and there are many!) must point to the same banks upon being read back in as they did when they were written out. This single block of data can then be handled without much further processing by Jazelle when being written out or read back in. (Jazelle does break it up so that tape record lengths are obeyed, but little else is done).

Each bank can be packed in a different way, given different amounts of actual information stored in its variables. Each bank benefits from not having its Jazelle header stored along with the bank, although the headers must sometimes be reconstructed individually upon readin. The bank to which the most attention was paid is MCPART, owing in large part to the fact that it is by far the most numerous

bank on the list above (for MC events, at least!), and because it is one of the banks more easily packed. Here's its template sans comments:

```
Bank MCPART Context=MCEVENT Maxid=5000 "MC Particle parameters"
  Real    P(3)        "X,Y,Z momentum of particle at track origin"
  Real    E           "Energy of particle"
  Real    PTOT        "Total momentum at track origin"
  Partid  PTYPE       "Particle type"
  Real    CHARGE      "Charge of particle"
  Bits    ORIGIN      "Where did this particle come from/go to" ! Note 1
  Real    XT(3)       "X,Y,Z of termination"
  Key     PARENT-->MCPART "Key giving parent particle"
EndBank
```

Many improvements can be had here. For one thing, PTOT=SQRT(P(1)**2+P(2)**2+P(3)**2), and need not be stored. Another obvious one is that the energy E can be calculated from PTOT as long as the particle has a usual mass. Most MCPARTs correspond to rather ordinary paricles -- photons, pions, kaons, protons, electrons,... One need only look up the mass using PTYPE and reconstruct E=SQRT(PTOT**2+MASS**2). This does not work for all particles, such as wide resonances or particles whose masses are likely to be determined better at a later date. So the list of "ordinary" particles whose mass is assumed by the compressor is:

photon
neutrino
electron
muon
pion (charged and neutral separately)
kaons (charged and neutral separately)
proton
neutron

The values for their masses are taken from PDG '92. Needless to say, a bit must be reserved in some word somewhere to say whether the energy E has been compressed out or not. The charge does not need 4 bytes for a real value. Only nine different charges of particles are really ever needed: 0, +-1/3, +-2/3, +-1, +-2 (alpha particles are fairly common). One can get away with only 4 bits here. The maximum bank ID is 5000, and can fit in 13 bits (maxiumum 8192). That way, one I*4 word can contain the MCPART bank ID, the bank ID of its PARENT, and still have room left over for the charge and that bit to tell us if we've squeezed out the energy. Most PTYPEs are small integers, but there is no real limitation on them. Sometimes particles are added to our listings by generators and decay packages that pick new regions of PTYPE numbering (>1 million for example) so as not to clash with other PTYPEs. The origin bits and the termination point do not compress.

Keys

Each key has three pointers in it, DATA, FIRST, and SAME. These implement Jazelle's relational table structure, which can be traversed using the TABLELOOP constructs in IDA and PREPMORT. The only pointer with any information in it is the DATA pointer; the other two are provided for convenience in finding related MCPARTs with either the same PARENT.DATA or different PARENT.DATA pointers. The SAME pointer is provided so one may loop through MCPARTs sharing a common parent simply by chasing SAME pointers. The FIRST pointer is provided so that if one loops through MCPARTs following the FIRST pointers, one finds a different parent each time, which speeds the search for a particular parent, if you don't have to loop through lots of sibling particles you aren't interested in. Pictorially, it looks like Figure 1.

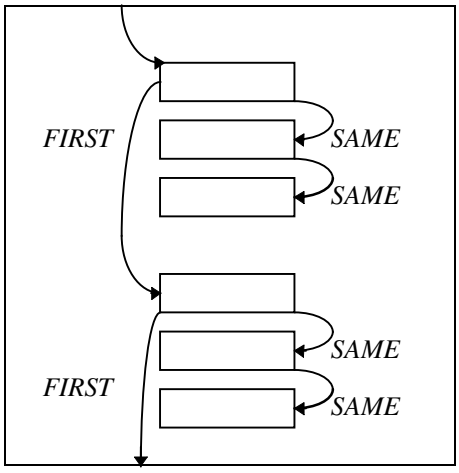

Figure 1. Jazelle relational table structure.

The hard part is that the natural ordering of the banks imposed by the keys is different from the numerical ordering of the bank Ids. Jazelle can add banks to the end of a list conveniently because it keeps track of a pointer to the last bank in a list, and can wire the JB$FORPT and JB$BAKPT pointers in the header easily. To insert a bank into the middle requires a search through all banks in the family to find where the new bank belongs in order to wire the forwards and backwards pointers together. The same is true of keys. Writing banks out in sequential order potentially scrambles the order of the keys. Searches through the existing list of banks to find another bank with the same parent have to be made.

The solution is to write the banks out in the natural order that the key imposes, so that the SAME and FIRST pointers may be filled in from the ordering context, rather than being stored in the compressed record. To speed bank addition, a low-cost version of the bank adder is called, JBADDN, which does not fill in the forwards and backwards pointers. The banks must be sorted at the end and their forwards and backwards pointers filled in, but this task is very easy because each bank already comes with its own ID which specifies its position in the final sort order. A convenient self-sorted list is provided by the indexed pointer facility, and so MCPART is JZPIDX'd by the decompressor. Then to fill in the JB$FORPT and JB$BAKPT, one need do a single loop over the indexed pointers and fill these in. This way, there are no $O(n^2)$ operations anywhere in the decompressor, and the indexed pointers are provided for free for the user.

A bank with two keys (MCPNT, for example) can be handled analogously. Most of the bank (including the first key) is written out in the natural order imposed by the first key. Then a second list of numbers is written to the compressed record containing MCPNT ID's and the second key's bank pointer's ID (two I*2's to save space), written in the natural order of the second key. Then the second key's FIRST and SAME pointers can be filled in by ordering context also. After both keys are filled in, the indexed pointer table is traversed to fill in JB$FORPT and JB$BAKPT.

Another Example Bank -- PHPOINT

Another bank which compresses nicely is PHPOINT. It also aids in the compression of other banks.

```
Bank PHPOINT Context=DST Maxid=1000 "Pointers to Particle Information"
  Pointer PHPSUM --> PHPSUM "Pointer to PHPSUM bank "
  Pointer PHCHRG --> PHCHRG "Pointer to tracking information"
  Pointer PHKLUS --> PHKLUS "Pointer to calorimetry information"
  Pointer PHKELID --> PHKELID "Pointer to calorimetry hypothesis"
  Pointer PHWIC  --> PHWIC  "Pointer to muon detector information"
  Pointer PHCRID --> PHCRID "Pointer to crid information"
Endbank
```

It already can be seen that bank ID's must be stored instead of pointers, and that these take only I*2's. But further compression can be had here. The PHPOINT bank always has the same bank ID as the PHPSUM it points to. While this is not assumed, it is used whenever it is true, and it is flagged with a bit. When true, the pointer is not written to the compressed record, and the decompressor copies the pointer to the corresponding PHPSUM. Most of the rest of the pointers are usually zero, usually with one or two exceptions. So a bit mask is kept (in the same word as the PHPOINT bank ID and that bit for PHPSUM), to tell the decompressor which pointers have been stored. These are then packed in as I*2's only if they are nonzero. This bank comes in the record after all of the subsystem banks it points to have been read in, so that real pointers may be put in their place. Some banks have redundant pointers. For example, PHCHRG has a pointer to PHPOINT. This pointer is not written out at all, and when PHPOINT gets read in, the corresponding PHCHRG gets its PHPOINT pointer filled in when PHPOINT's PHCHRG pointer is found. The indexed pointer tables are invaluable for quickly finding a bank pointer when a bank ID is

known. An assumption here is that if a PHCHRG's PHPOINT pointer is nonzero, then the corresponding PHPOINT bank exists and has a consistent PHCHRG pointer pointing back. If this is not the case, then these pointers may not compress properly. Because pointers are stored as bank ID's, it is impossible to change the bank type of a pointer without changing the compressor and decompressor code, as happened when PHKTRK was removed from PHPOINT and PHKELID was put in its place.

Listing of Known Assumptions

In order to take advantage of compressable features of the MiniDST, several assumptions are made about the redundancy of the data. The ones involving MCPART are already described above, along with their reasons for their existence. Some assumptions affect the ability of the compressor to compress the banks, and others simply are imposed on the decompressed record, regardless of whether or not the hold in the data before compression.

1) MCPART bank ID < 2^13, or 8192, in order to compress the ID, the parent ID, and the charge in the same I*4.

2) MCPART%(CHARGE) is one of the following values: 0, +-1/3, +-2/3, +-1, +-2, +-3, +-4, +-5. Particles with other charges will not be reported properly upon decompression.

3) The components of MCPART%(P) add in quadrature to yield MCPART%(PTOT). This does not hold true for particles of type 98, GEANT shower axes. While this does not create an error condition, the decompressed banks will satisfy the relation between P and PTOT.

4) The masses of the photon, neutrino, charged pion, neutral pion, charged kaons, neutral kaons, proton, neutron, electron, and muon are all as given in the 1992 PDG book.

5) Pointers, when nonzero, point to actual banks. A violation of this assumption when the first version of NUKE98 was made caused the compressor to crash on an access violation. Even so, it is a good practice to stick to this rule, since much SLD code would crash similarly if dangling pointers were left around.
The reason for this assumption is that pointers are stored in the compressed record as bank ID's. In order to find the bank ID, the bank pointed to must be present when the record is written out. Sometimes Jazelle will delete a bank pointing to a bank which has been deleted (this happens with MCPART), but test this feature on the banks you are deleting to make sure that other banks with dangling pointers either are deleted or have their pointers zeroed.

6) Pointers point to the kinds of banks they say they do. Putting a pointer to a PHKLUS bank into PHPOINT%(PHCHRG), for example, will cause the compressor and decompressor to behave in an undefined manner (likely just crash the program at some point, but perhaps with other mysterious side effects that are hard to track down).

7) Templates defining the banks listed above do not change. If anything besides a comment changes in one of the banks listed, please check the compressor code for ramifications. Changing the length of a bank is particularly noisome, as the compressor often copies the bank by its length which is sometimes found from the template, and sometimes hardwired in the compressor code (depending on whether something in the middle of the bank could be squeezed out or not). Even an "innocuous" change, such as redefining an integer to a real, even if the value was previously unused, will produce undefined behavior in the compressor, largely due to hardwired platform-conversion rules which assume the type of every element in every bank. Likewise do not swap an I*4 for two I*2's. This could cause realignment of data and platform-conversion errors (It has happened!). If a template must change, then the appropriate portion of the compressor, decompressor, and converter must be updated. Often the change can be made with transparent backward compatibility with existing datasets, but sometimes it is necessary to increase the version

number, stored in the table of contents, to be used as a switch to tell the converter and decompressor to handle the new as well as old formats.

8) CRID combined log likelihoods are the sum of the gas and liquid log likelihoods, with an optional normalization which is constant for the five particle ID's. If the CRID block RC's are less than zero, then the corresponding likelihoods will be set all to zero upon decompression (they will not be written out to the compressed record).

9) PHPOINT pointers are present (when the banks they need to point to are present), consistent, and "unique." For example, for each PHPSUM bank, there is a PHPOINT bank. The PHPSUM's bank points to the corresponding PHPOINT bank and the PHPOINT bank points back to the same PHPSUM bank. This assumption is necessary because at most one of the pointers is written out to the compressed record, and neither of them if the ID's are also the same. Two PHPOINTs should not point to the same PHPSUM, or to any other bank, for that matter. To express many-to-one or many-to-many relationships, relational tables should be used.

10) PHTRK is not in the compressed bank list, and so the PHTRK pointer in PHCHRG will be zero upon decompresson always, even if PHTRK banks are also added to the output record in uncompressed format. See the discussion at the top of this memo for adding banks to the output list when using compressed records.

11) With the proposed PHKHIT bank, there is at most one PHKHIT bank per PHKLUS bank, and there is exactly one PHKLUS bank per PHKHIT bank. PHKLUS banks can be without PHKHIT banks. These conditions are required because each PHKHIT will be stored along with its corresponding PHKLUS, and at most one will be stored with each PHKLUS bank.