A Maintainers Guide to MIDASsort

Duncan Appelbe, CCLRC Daresbury Laboratory

Table of Contents

Introduction 1
The GUI 2
- The Sort Control Frame. 3
- The Sort Settings Frame 5
- The MEMSAS Control Frame. 6
- The Sort Options Frame. 7
- The Debug Levels Frame. 8
- The View Data Frame. 9
- The User Variables Frame. 10
- The Tape File Info Frame. 11
- The Disk File Info Frame. 12

Control Programs 14
Sort Package Libraries. 15
- Sort-main 15
- Program flow 15
- Routines to read the data 16
- Routines to decipher the data 18
- Routines used during the sort 19
- Routines Available to the user 20

Writing Sort Programs 21
- Global Variables 21
- Using "C" 21
- Using "Fortran" 24

ToDo List 24

Revision History
Revision 0.01 Wed Jan 28 09:26:50 2004 dea
Changed the original layout to one slightly more logical.

Introduction

MIDASsort is intended to be a multifuction data sorting package for the MIDAS data Analysis/Acquisition suite that is available by http/ftp from the Daresbury Nuclear Physics Group. MIDASsort has been written to replace SunSort and CSort, both of which are based upon the OpenLook graphics package that is no longer supported by SUN. This package should *not* be confused with MTSort that is supported by Liverpool Nuclear Physics Group.

The GUI for MIDASsort has been written using the MIDAS Tcl libraries, and makes use of the MIDAS package for its control and to display different histograms.

This document has been written in xml and processed using FOP and the DocBook DTD. The source for this file and some of the associated scripts can be found in the documentation directory of the MIDASsort development tree.

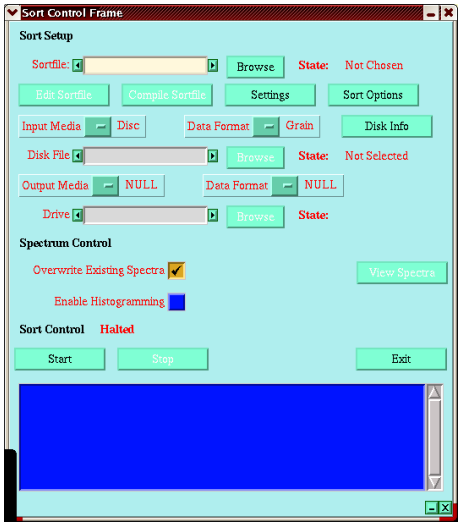
The individual elements of MIDASsort can be found in the directory `$(MIDASBASE)/MIDASsort`.

In order to use MIDASsort under Microsoft Windows you *MUST* have installed cygwin and the gcc compilers. The system path should be modified to include the `C:\cygwin\bin` directory.

The GUI

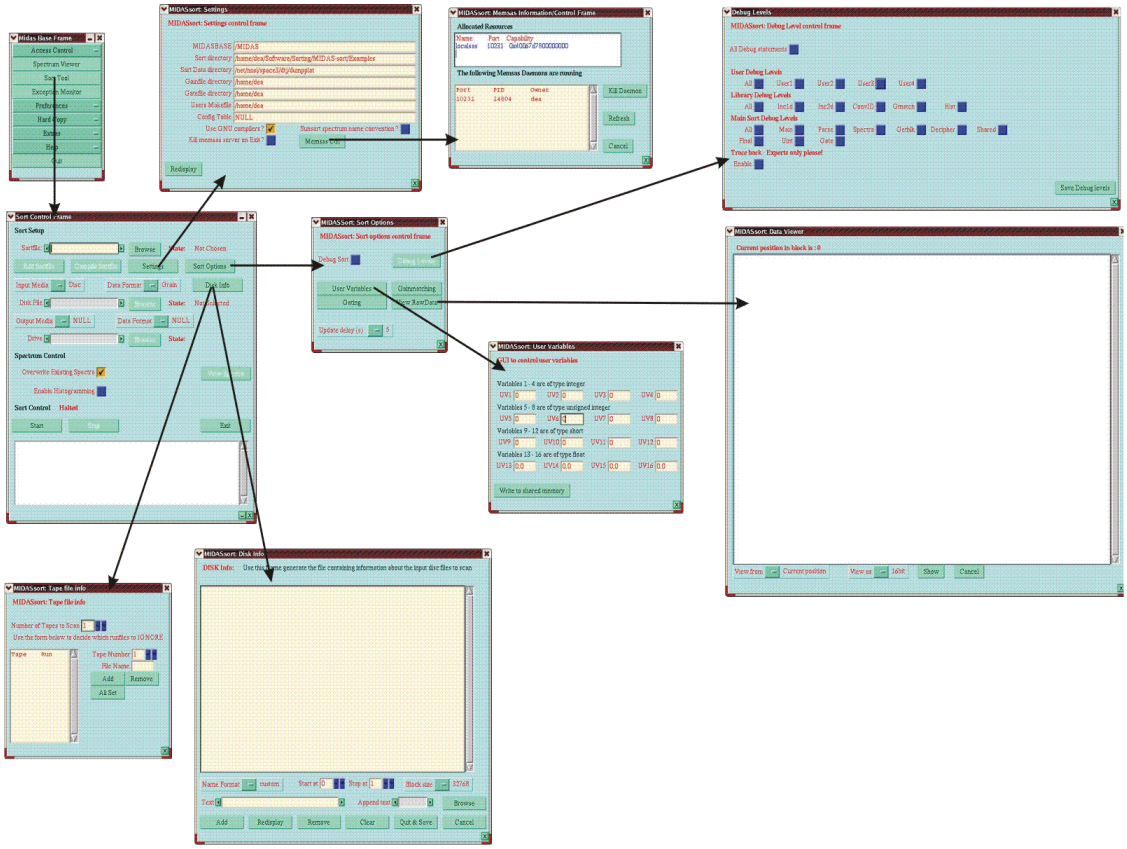
MIDASsort is controled through a GUI, this GUI is launched from the MIDAS base frame using the button "Sort Tool". Once initiated the software sets up the local sas resource that is required to view spectra, then starts the "Sort Control Frame", see below.

Figure 1. The Sort Control Frame



Once this GUI has been launched it can be used to control all aspects of the sorting mechanism. The relationship between the different frames are shown below:

Figure 2. The relationship bewteen the Sort Control Frame and its daughters



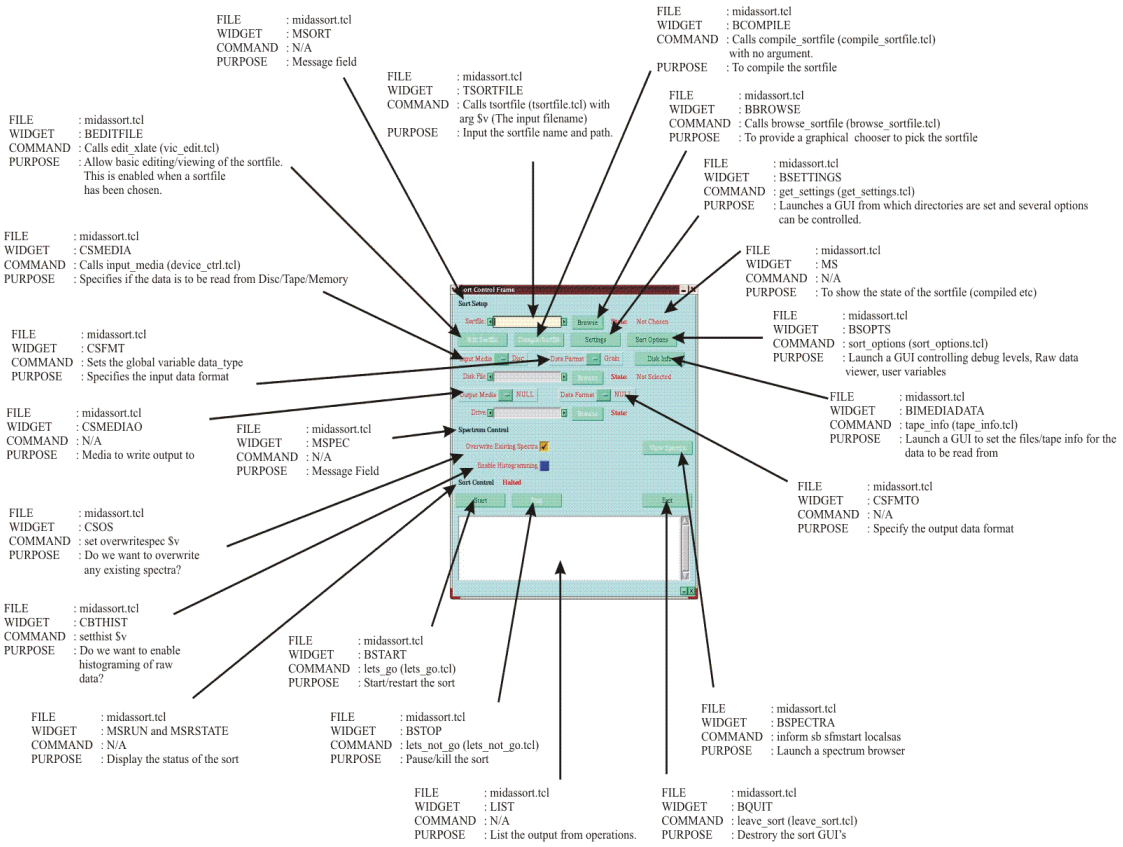
The tcl that is used to define the GUI Frames and the associated actions is split between numerous files. In the main the name of these files relates to the functions that it contains. The main frame is setup from the file `midassort.tcl` while all of the event handlers are defined and associated within the file `click.tcl`, while all of the widget "help" information is contained within the file `midassort.txt`.

In the following sections, details about the different GUI frames are discussed. When widgets are specified, they are the widgets defined by the MIDAS tcl library.

The Sort Control Frame.

For ease, the figure below shows the names and functions of the different widgets that compose the Sort Control Frame.

Figure 3. A synopsis of the widgets that compose the Sort Control Frame.



These widgets are listed below:

- MSORT: A message widget.
- TSORTFILE: A text widget used to input the path and name of the sortfile to use.
- BBROWSE: A button widget that launches a graphical file browser
- MS: A text widget that displays the compilation status of the sortfile.
- BEDITFILE: A button widget that launches an TCL window that allows the user to perform basic editing of the sortfile.
- BCOMPILE: A button widget that launches a C program that parses the input sortfile, extracts details of the spectra you wish to create then compiles the sortfile and links in the sort libraries. The parseing program writes tempory files to the `/tmp/tcl(pid)` directory.
- BSETTINGS: A button widget that launches the settings control frame.
- BSOPTS: A button widget that launches the sort options control frame.
- CSMEDIA: A choice-stack widget that allows the user to specify the input data stream. Currently the available input data streams are "disk", tape, or using the shared memory buffer of the MIDAS tape server.
- CSFMT: A choice-stack widget from which the event decoder is selected.
- BIMEDIADATA: A button widget that launches a control frame allowing the user to specify the Volume/Filenames for either the input tapes or disk files.
- CSMEDIAO: A choice-stack widget that can be used to specify the output device for data.
- CSFMTO: A choice-stack widget allowing the user to specify the format of the data to be written out.
- MSPEC: A message widget.
- CSOS: A check-box widget. If selected all spectra will be overwritten, else they will be added to.
- CBTHIST: A check-box widget. If ticked Histograming will be enabled (requiring a configuration file). Histogramming works with all data formats *except* the GREAT event handler.
- BSPECTRA: A button widget that launches a spectrum browser.
- MSRUN: A message widget.
- MSRSTATE: A message widget displaying the current status of the sort program.

- BSTART: A button widget that starts/continues the sort program.
- BSTOP: A button widget that pauses/stops the sort program.
- BQUIT: A button widget that destroys the control frame and all visable daughters, then saves the settings before exiting.
- LIST: A galley widget that displays the messages from some of the programs called by this GUI.

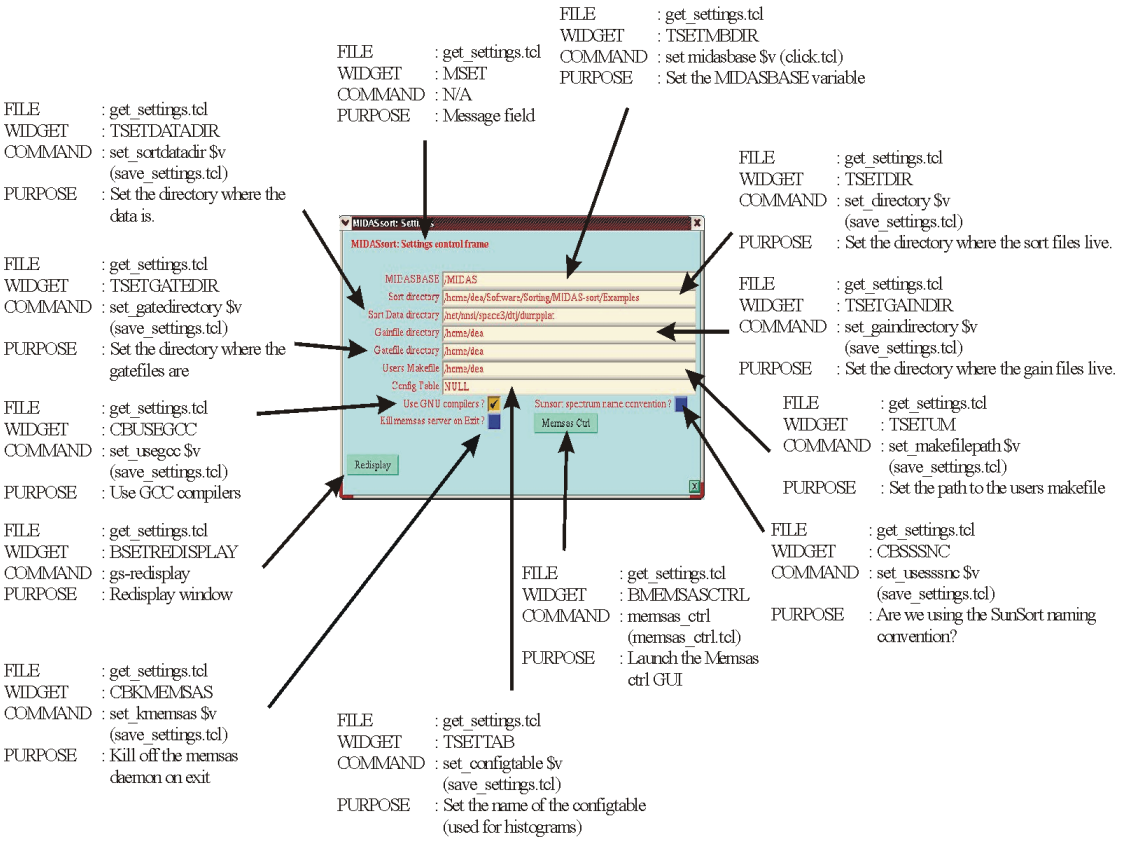
The Sort Settings Frame

This GUI is used to specify the locations/directories in which certain files exist: ranging from the sortfile directory, to the directory where disk files containg data are kept. In addition it is possible to choose between GNU and "other" compilers, whether you wish to use the SunSort naming convention, and if the spectrum memory server should be killed off when the Sort tool is exited from.

It should be noted that all of the variables controlled by this GUI are saved as preferences in the file `$(HOME)/.midas/preferences`.

For ease, the figure below shows the names and functions of the different widgets that compose the Sort Settings Frame.

Figure 4. A synopsis of the widgets that compose the Sort Settings Frame.



These widgets are listed below:

- MSET: A message widget.
- TSETMBDIR: A text widget that shows where the MIDASBASE enviroment variable is set to, changing the entry in this field does not do anything.
- TSETDIR: A text widget allowing the directory where the sortfiles live to be set. This variable is

used by the BBROWSE widget on the Sort Control Frame.

- TSETDATADIR: A text widget where the directory containing the data to be sorted can be specified. This variable is used in the "disc" frame.
- TSETGAINDIR: A text widget specifying the directory where any gain files reside. This feature is not currently implemented.
- TSETGATEDIR: A text widget specifying the directory in which any GATEFILES reside. This feature is not currently implemented.
- TSETUM: A text widget where the users Makefile can be specified. This makefile is included in the makefile generated at compilation time.
- TSETTAB: A text widget used to specify the users configuration table. This Configuration table is used when data is to be histogramed.
- CBUSEGCC: A check-box. If ticked gcc compilers will be used, else the makefile will assume that "cc" and "f77" are valid compilers.
- CBSSSNC: A check-box widget used to specify if the sunsort naming convention is used. This really only affects lists of spectra that are differentiated only by number. In Sunsort this number is the spectrum id number. If this widget is not ticked then the spectrum name numbering will start at 0.
- CBKMEMSAS: A check-box used to determine if the memsas daemon that is associated with this sort will be killed when the sort package exits. (The memsas still remains if the Sort Control frame is displayed/minimised).
- BMEMSAS_CTRL: A button widget that launches the memsas_ctrl frame. This GUI is used to show if any daemons are running on your machine, and their associated ports.
- BRESETDISPLAY: A button widget that forces a redisplay of all the entry points in this frame.

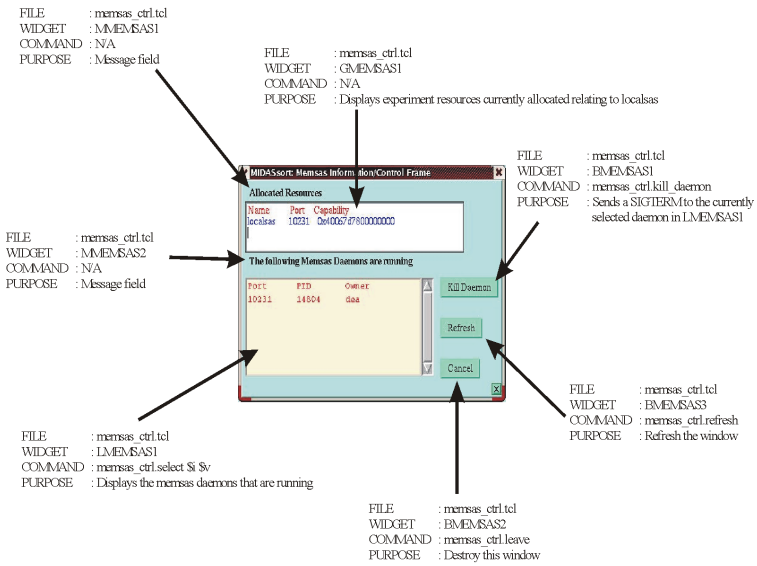
To kill the frame the [x] has to be used

The MEMSAS Control Frame.

In order to sort spectra using this package a memsas daemon must be running. When the Sort Control Frame is launched it is determined whether any daemons are running, and if any resources (localsas) are allocated. If not a daemon is started, this frame displays the running daemons and the allocated resources. It is also posible to kill off memsas daemons using this frame.

For ease, the figure below shows the names and functions of the different widgets that compose the Sort Settings Frame.

Figure 5. A synopsis of the widgets that compose the Sort Memsas control Frame.



These widgets are listed below:

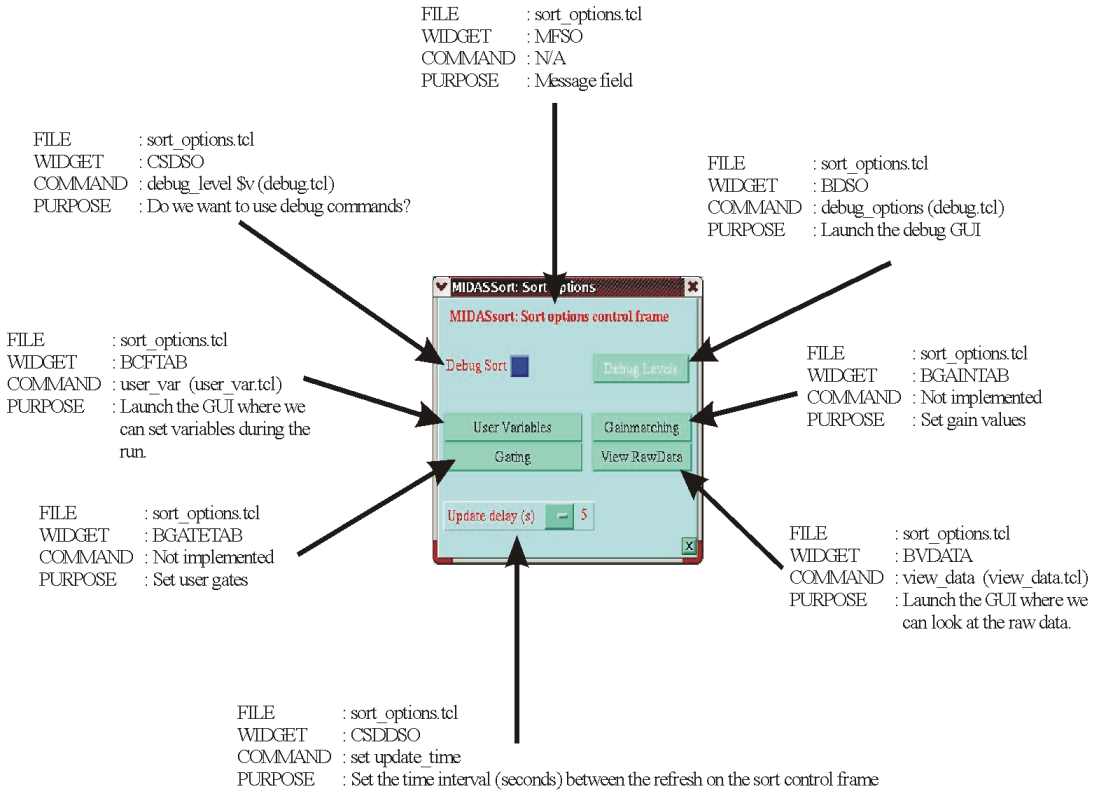
- MMEMSAS1: A message widget.
- GMEMSAS1: A gally widget used to display the "localsas" resources currently allocated to this session. The user can't modify any of the values displayed directly.
- MMEMSAS2: A message widget.
- LMEMSAS1: A list widget, used to display all of the memsas daemons currently running on the host machine. The daemons are identified by portnumber (10230-10249), process ID and the owner. Individual daemons can be selected by clicking on them.
- BMEMSAS1: A button widget used to send a SIGTERM signal to the selected daemon, thus killing it, and removing any spectra stored in its shared memory.
- BMEMSAS2: A button widget that destroys this frame.
- BMEMSAS3: A button widget that forces a redisplay of the data shown in the frame.

The Sort Options Frame.

As the sort runs there are several options that users would wish to have. These options are controled from the sort options frame. These options make use of shared memory and can be set before a sort is started or as the sort runs.

For ease, the figure below shows the names and functions of the different widgets that compose the Sort Options Frame.

Figure 6. A synopsis of the widgets that compose the Sort Options Frame.



These widgets are listed below:

- MFSO: A message widget.
- CSDSO: A check-box widget. If ticked the BDSO widget becomes active, allowing debugging to be enabled.
- BDSO: A button widget that launches the debug levels frame.
- BCFTAB: A button widget used to launch the user_var frame, from which the user can change variables as the sort runs.
- BGAINTAB: A button widget, not implemented at present.
- BGATETAB: A button widget that is not implemented.
- BVDATA: A button widget that launches the view_data frame, from which the user can see the raw, unformatted data in the vicinity of the current block position.
- CSDDSO: A choice-stack widget, that allows the user to set the update interval for the sort control frame.

The Debug Levels Frame.

Given the compelxity of this sort package it is difficult to debug individual elements. The released version is as free of bugs as is possible, but bugs will still be present. In order to aid in any debugging process there are a number of debug levels (for want of a better word) available. When enabled these "levels" will print out data to stdout which can be checked to see if the software is performing as expected.

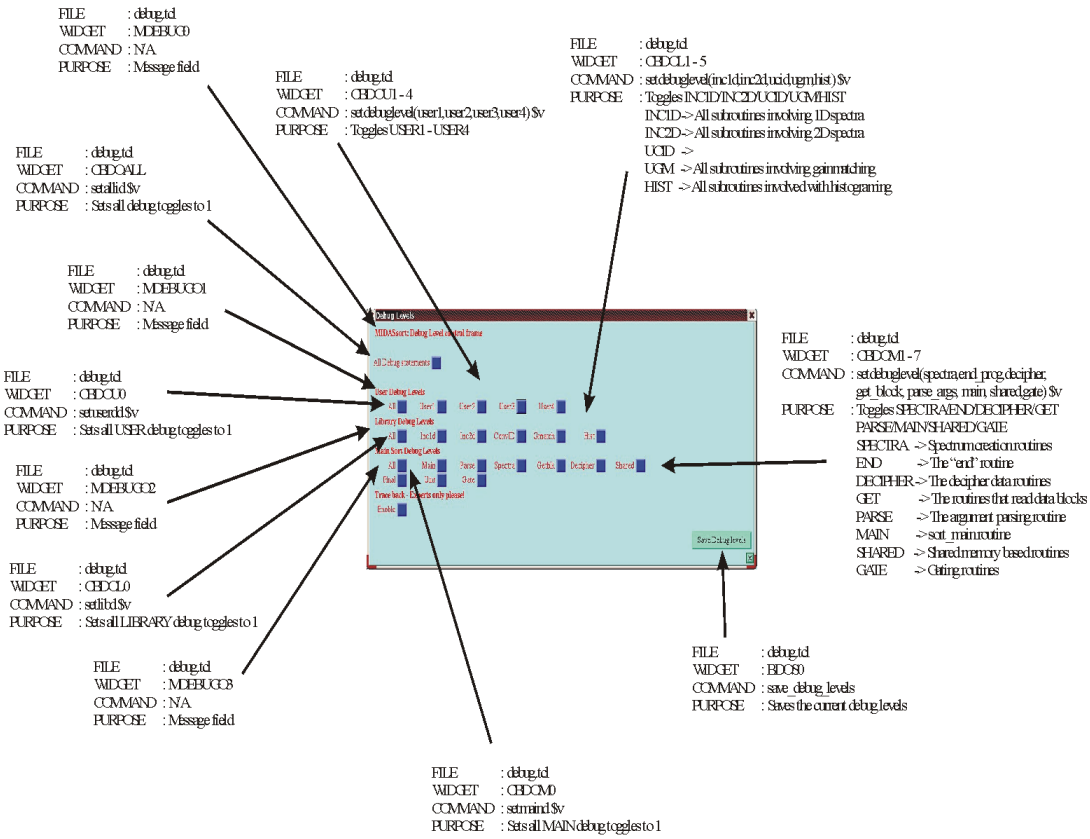
These debug levels are activated via this GUI and use shared memory. To use them in your code the following template should be used:

```
if ( ( *debug_level & GATE ) == GATE )
 {
  (void *) printf("gate1d : In 1D gating routine called with\n"
                  "       : data = %i\n", data
  );
}
```

Where the user would substitute "GATE" for one of the other defined variables. These different variables are discussed later in this section. However as far as the end user is concerned there are four debug toggles available: USER1 USER2 USER3 and USER4.

For ease, the figure below shows the names and functions of the different widgets that compose the Debug Levels Frame.

Figure 7. A synopsis of the widgets that compose the Debug Levels Frame.



These widgets are listed below:

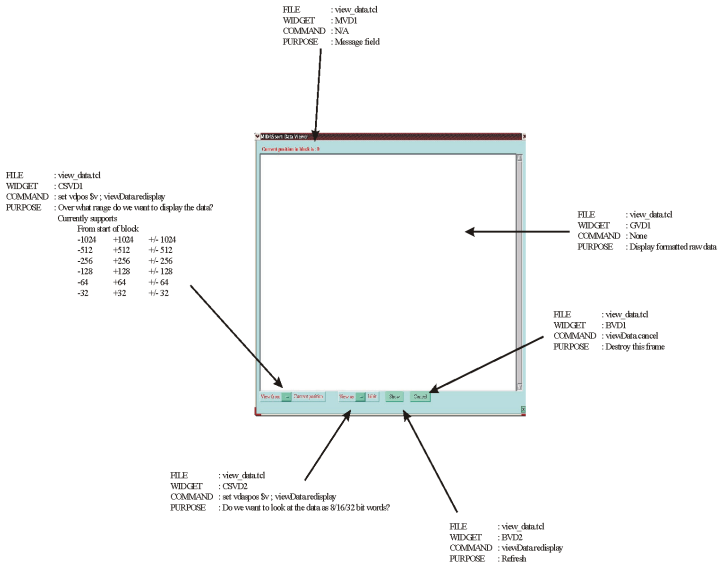
- MDEBUG0: A message widget.
- CBDOALL: A check-box widget. If toggled this will set all of the available debugging options on.
- MDEBUG01: A message widget.
- CBDOU0: A check-box widget. If ticked this will set all user debug toggles to on.
- CBDOU1(2,3,4): Four check-box widgets that are used to toggle USER1 - USER4 respectively.
- MDEBUGO2: A message widget.
- CBDOL0: A check-box widget. If ticked all of the Library debug levels are set.
- CBDOL1(2,3,4,5): Five check-box widgets that are used to toggle INC1D (all subroutines involving 1D spectra), INC2D (all subroutines involving 2D spectra), CONV1D, UGM (all subroutines involving gainmatching)and HIST (all subroutines involving histograming).
- MDEBUGO3: A message widget.
- CBDOM0: A check-box widget. If ticked all of the MAIN debug levels are set.
- CBDOM1(2,3,4,5,6,7): Seven check-box widgets that are used to toggle SPECTRA (spectrum creation routines), END (the End routine), DECIPHER (the decipher_data routines), GET (the routines that read the data), PARSE (the argument parsing routine), MAIN (the sort_main routine), SHARED (shared memory routines) and GATE (gating routines).

The View Data Frame.

This frame is used to allow the user to look at the raw data before it has been unravelled by the event decoder routine. In order to obtain the data the user provides two variables: the first determines the range of data to be viewed, the second the format to look at the data (8/16/32bit words). This information is then passed to the program midasSortfb as arguments, which returns the current position in the data block and a stream of data to be painted to the screen. The data word at the current position in the block is highlighted in red.

For ease, the figure below shows the names and functions of the different widgets that compose the View Data Frame.

Figure 8. A synopsis of the widgets that compose the View Data Frame.



These widgets are listed below:

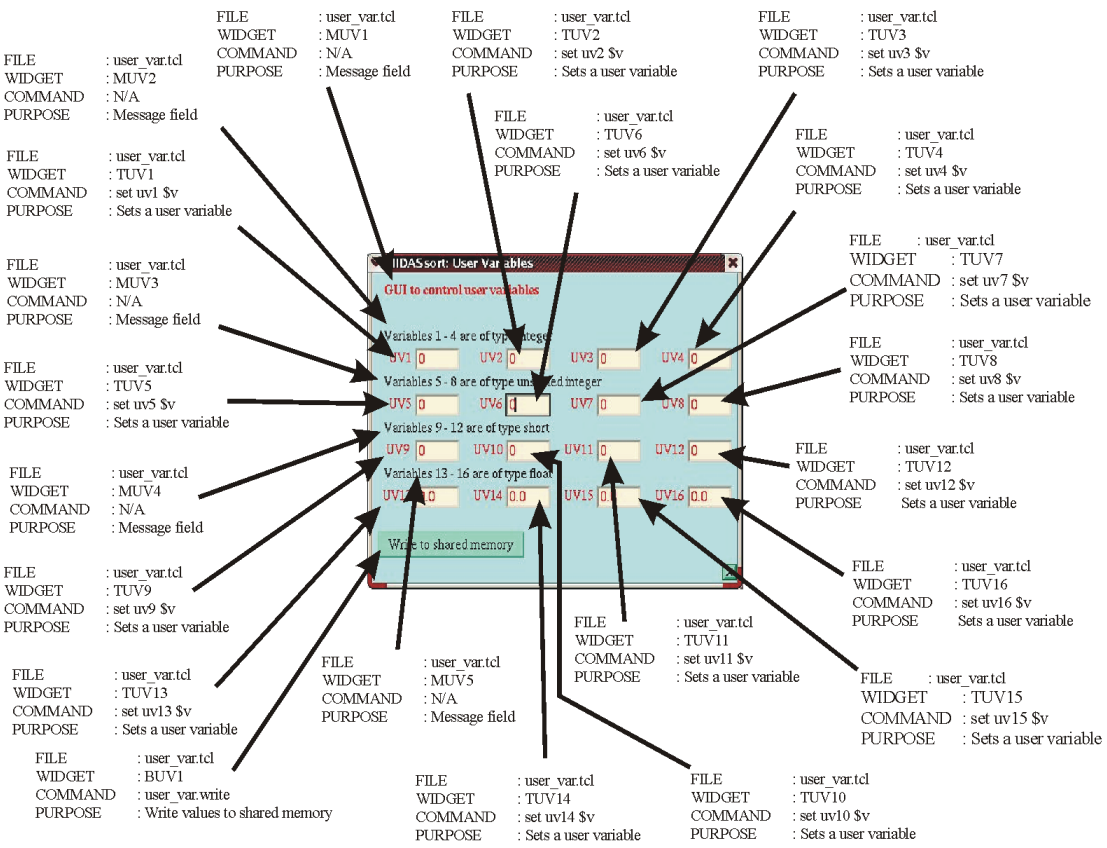
- MVD1: A message widget, used to display the current position in the block. Note that this is the actual position as it is assumed that all data words are 16bit prior to decodeing.
- GVD1: A Galley widget that is used to display the raw data. The user can not modify anything in this widget.
- BVD1: A button widget used to destroy this frame.
- CSVD1: A choice-stack widget used to determine the range over wich the data should be displayed. This range is with respect to the current position. If the choice goes out of bounds the bounds are altered automatically.
- CSVD2: A choice-stack widget that is used to specify the number of bits per word.
- BVD2: A button widget, used to force a redisplay of this window. This window is also redisplayed if either CSVD1 or CSVD2 are altered.

The User Variables Frame.

Occasionally the user may want to have the ability to change some variables as the program is running. While this is not recomended, this option has been incorperated into MIDASsort. These variables can be set/modified by means of the "User Variables" frame. Four different types of variable can be used, int, unsigned int, short int and float. The variables are written to shared memory either before the sort starts or as the sort progresses. These variables are only written to shared memory when the "Write to Shared Memory" button is pressed - thus the numbers that are displayed may not reflect the actual variables.

For ease, the figure below shows the names and functions of the different widgets that compose the User Variables Frame.

Figure 9. A synopsis of the widgets that compose the User Variables Frame.



These widgets are listed below:

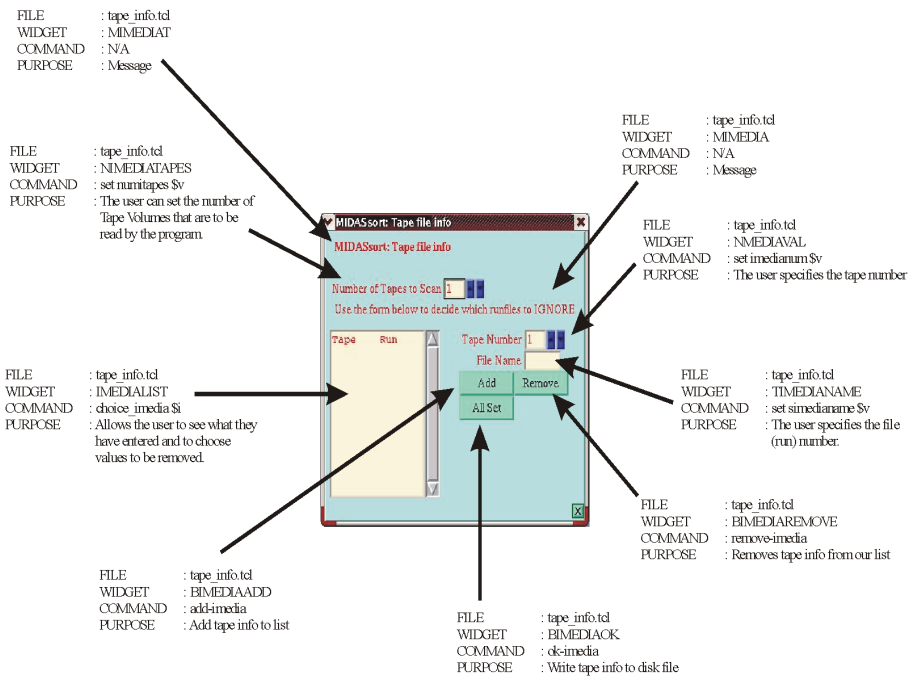
- MUV1(2,3,4,5): Message widgets.
- TUV1(2,3,4): Text widgets that are used to input/display the integer variables.
- TUV5(6,7,8): Text widgets that are used to input/display the unsigned integer variables.
- TUV9(10,11,12): Text widgets that are used to input/display the short integer variables.
- TUV13(14,15,16): Text widgets that are used to input/display the float variables.
- BUV1: A button widget that is used to initiate the writing of these variables to shared memory.

The Tape File Info Frame.

If the data to be sorted is contained on tape the user needs, at a minimum to specify the number of tapes that are to be read. In addition if the tapes in question are ANSI format then information about the file-name is required. MIDASsort assumes that the filenames are different, so tapes are identified via a number. Once all of the required data has been input it is saved to the file `tape.info` in the tmp directory associated with current MIDAS session.

For ease, the figure below shows the names and functions of the different widgets that compose the Tape File Info Frame.

Figure 10. A synopsis of the widgets that compose the Tape File Info Frame.



These widgets are listed below:

- MIMEDIAT: A message widget.
- NIMEDIATAPES: A number widget, used to specify the total number of tapes to be read.
- MIMEDIA: A message widget.
- IMEDIALIST: A list widget used to display tape information. Individual items can be selected from this list.
- NMEDIAVAL: A number widget used to specify the tape number to be assigned to the current file name.
- TIMEDIANAME: A text widget used to specify the name (6 chars) of a file to be read.
- BIMEDIAADD: A button widget, used to add tape info to the list.
- BIMEDIAREMOVE: A button widget used to delete the selected tape file from the list.
- BIMEDIAOK: A button widget that writes the data to disk and exits this frame.

The Disk File Info Frame.

If the data to be sorted reside on a disk, then that information needs to be provided to the sort program. This is achieved through the "Disk Info" frame.

There are three options for specifying the names of files to be read:

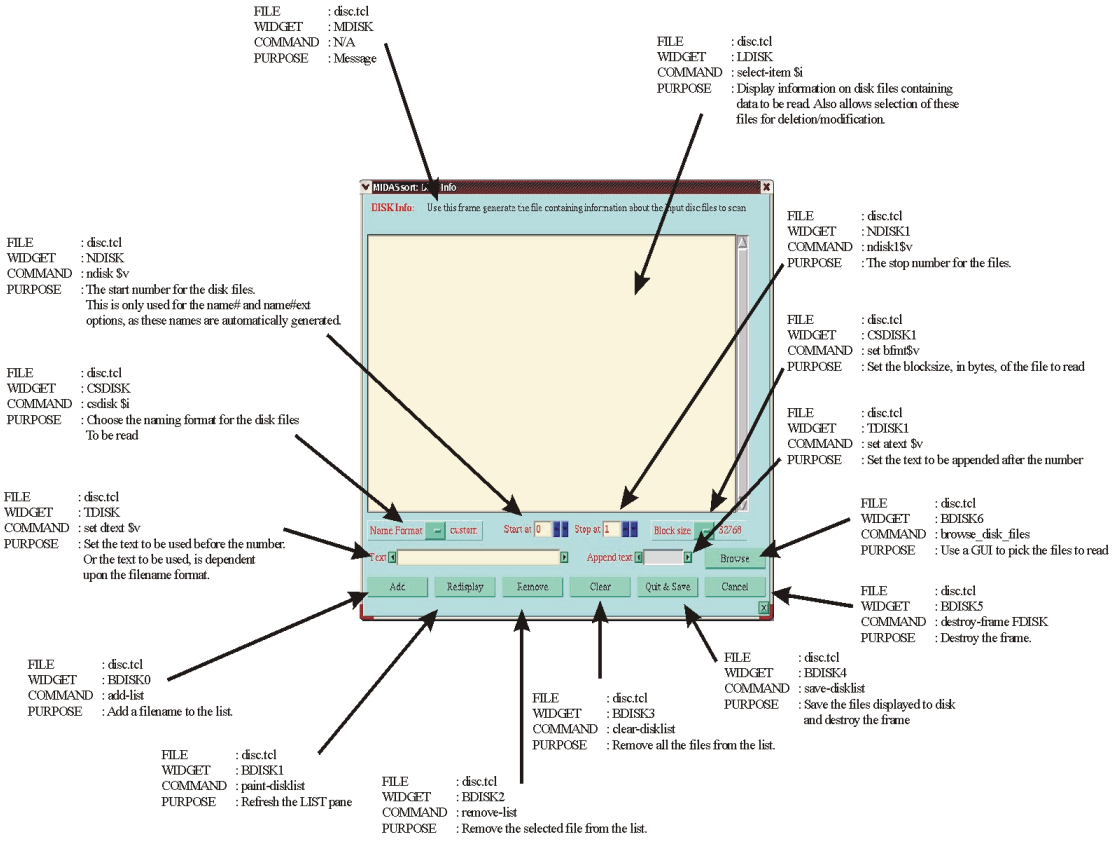
i. Multiple files with names in the format name#.extension, where "#" represents a number.
ii. Multiple files with names in the format name.extension#, , where "#" represents a number.
iii. Individual files.

In the first two cases the individual text can be specified, as can the start and stop numbers, the tcl frame will then generate the complete list of filenames. In the latter case names must be specified individually.

In addition the user must specify the block size, in bytes, of the disk file to be read. This will ensure that all of the data is sorted, in the majority of cases this will be 16384.

For ease, the figure below shows the names and functions of the different widgets that compose the Disk File Info Frame.

Figure 11. A synopsis of the widgets that compose the Disk File Info Frame.



These widgets are listed below:

- MDISK: A message widget.
- LDISK: A list widget used to display the information relating to the disk files to be read. If the file `disc.info` is present then the contents of this file are added to the list. It is possible to select the different disk files in this widget.
- CSDISK: A choice-stack widget used to specify which of the filename formats is being used to specify the files to be read (see above).
- NDISK: A number widget used to specify the *start* number for files using formats i or ii. The value in this widget is not used for the "custom" format.
- NDISK1: A number widget used to specify the *stop* number for files using formats i or ii. The value in this widget is not used for the "custom" format.
- CSDISK1: A choice-stack widget, used to specify the size of the blocks to read. The block size is measured in bytes.
- TDISK: A text widget that is used to specify the first part of the filename (formats i and ii) or the entire filename (format iii). The user can either type the filename directly into this widget, or can use the "Browse" button to pick the file graphically. If this latter option is chosen, and formats i and ii are specified the user will need to edit this text before "adding" to the list.
- TDISK1: A text widget that is used to specify the text to be appended to the filename. This widget is only enabled for formats i and ii.
- BDISK6: A button widget that is used to start a graphical browser to pick filenames.
- BDISK0: A button widget, used to add files to the list. If formats i and ii are specified then the files between *start* and *stop* will be created.
- BDISK1: A button widget, used to redisplay the list.
- BDISK2: A button widget that is used to remove the currently selected filname from the list.

- BDISK3: A button widget, used to clear all files from the list.
- BDISK4: A button widget used to save the data to the disk file `disc.info` in the current tmp directory. Once this has been competed the frame is destroyed.
- BDISK5: A button widget, used to destroy the frame without saving any changes to file.

Control Programs

Although much of the program control is handled within the Tcl code several "C" programs have been written that are started by the GUI. The source code for these programs reside in the `MIDASsort/Utilities` directory.

The programs currently used by MIDASsort are:

i. `compile_midas_sort`: This program is used to parse the sort file, generate the Makefile `make.midas_sort` and run the make routine.

ii. `filestatus_midas_sort`: This program is used to check if the sortfile is compiled, if the source is newer than the executable and if the configuration files are present in the tmp directory. The argument passed to the program is the name (and path) of the source file. The output from the program is specified as:

   - 0: No executable file is present.
   - 1: The program is compiled.
   - 2: The executable is older than the source.
   - 3: There is an executable file, but no configuration files are present in the tmp directory.

iii. `midasSortfb`: This program is used while the sort is in progress to extract the current position in the data block and the raw data. This information is stored in shared memory. The arguments that are passed to the program are range (how much data is to be displayed), wrdbits (display the data is 8/16/32 bit words), action ((r)ead/(w)rite/(d)elete). There are two outputs from the program: the first is the current position in the block (written to the file `msdata.txt`) and a stream of data to stdout (captured by the tcl).

iv. `midasSortshm`: This program is the principle method of accessing the shared memory components used by the sort program. The arguments to this program are action((r)ead/(w)rite/(d)elete), variable id (1-21) and value. The numbers associations are:

   1. : action - pause/restart/kill the sort.
   2. : debuglevel - what debug options are set.
   3. : toggle_hist - enable histograming.
   4. : user variable int
   5. : user variable int
   6. : user variable int
   7. : user variable int
   8. : user variable unsigned int
   9. : user variable unsigned int
   10. : user variable unsigned int
   11. : user variable unsigned int
   12. : user variable short int
   13. : user variable short int
   14. : user variable short int
   15. : user variable short int
   16. : user variable float
   17. : user variable float
   18. : user variable float
   19. : user variable float

v. `return_exe`: program that returns the name of the executable file. The argument is the source file.

vi. `tapestatus_midas_sort`: program that interogates the named tape drive, returning a string relating to its status:

- Exabyte 8500: Ready - An exobyte tape drive is ready for use.
- DLT: Ready - A DLT tape drive is ready for use.
- DLT1: Ready - A DLT1 tape drive is ready for use.
- Tape Ready - An unknown tape drive is ready for use.
- No such device - the chosen device is not connected/powered up.
- No Tape Loaded.
- Permission denied - you do not have permission to acces this drive.
- Device busy - The device is in use/performing an operation.
- Device error.

vii. `ts_status_midas_sort`: This program checks for the presence of a shared memory segment with the id of the MIDAS TapeServer shared memory. Note the program does not test that the TapeServer is running! There is no argument. The output from this program is:

- No Connection : The shared memory does not exist.
- Good Connection : The shared memory exists.

Sort Package Libraries.

The programs that make up the sort package library are split into two: those relating directly to the sort (found in `$(MIDASBASE)/MIDASsort/Sorting`) and those relating to the rest of the package (found in `$(MIDASBASE)/MIDASsort/Libs`)

Sort-main

The "main" routine is defined in the file `sort-main.c`. From this routine everything else is called, the order in which this is done is displayed in the next session.

Once the main routine has been entered the first action is to setup the signal handlers, followed by specifying that the routine "end_prog" should be called whenever the program exits. This is important as if an error forces the premature end of the sort the value "debug_status" is printed out. This variable changes as the program steps through in order to make it easier to determine the point at which the bug is found.

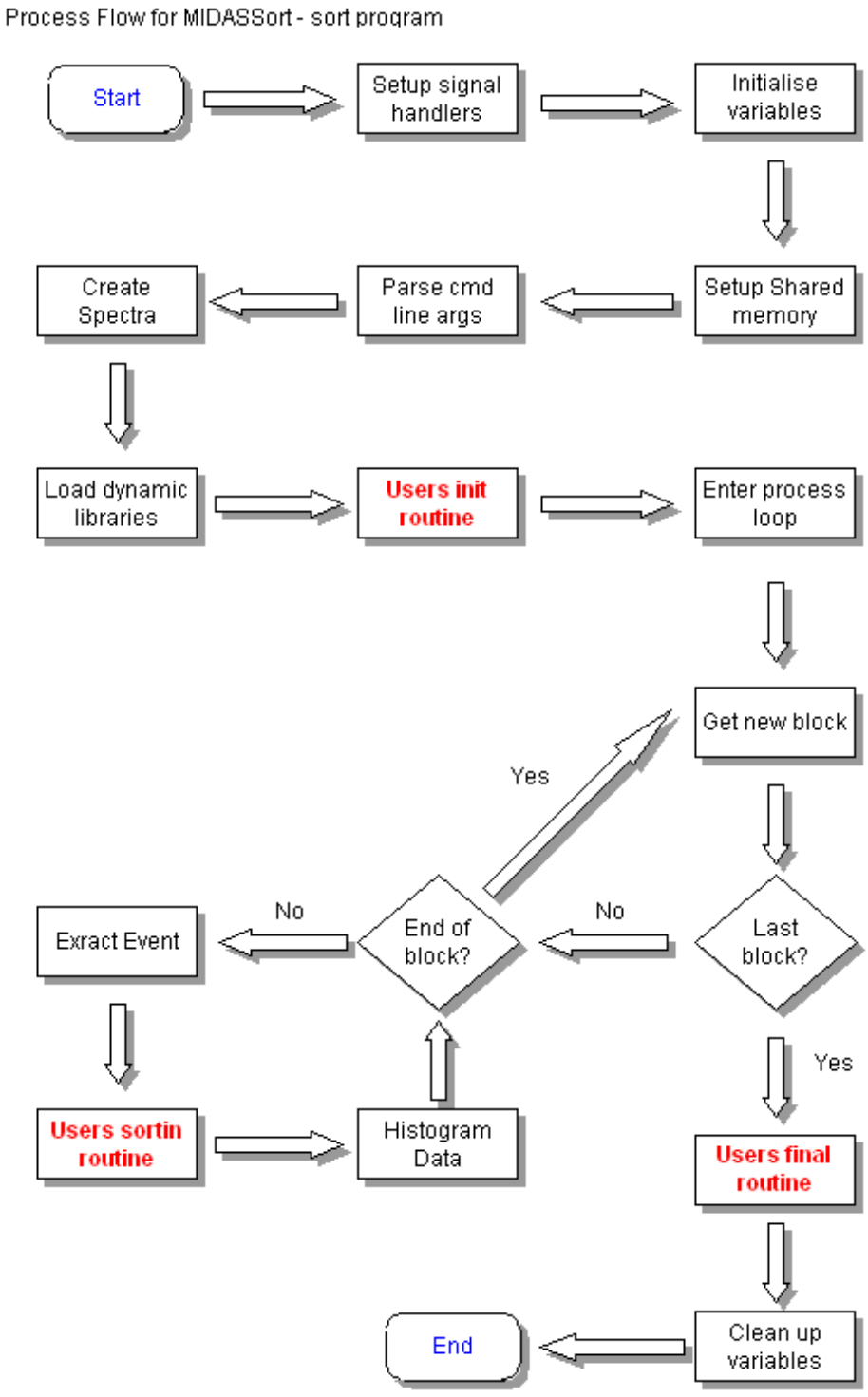
It should be noted, that as soon as the debug status has been changed via the GUI it takes imeadiate effect within the program.

Although the control of the program is handled via shared memory the state of the "action" variable is only checked at the end of each data block. So users should not be concerned if the program continues to run after it has been told to pause or stop.

Program flow

The figure below shows how the different routines relate to each other, and the interation between different routines.

Figure 12. Program flow datagram.



Routines to read the data

This section deals with the routines that read blocks of data into the program. At present three different routines are provided:

1. `tape` - This routine reads blocks of data from tape.

2. `disc` - This routine reads blocks of data from disk.
3. `midasTS` - This routine reads data from the shared memory allocated by the MIDAS TapeServer.

These routines are written as modules that are loaded dynamically at runtime. The reason for this is to allow the same sort program to access data from different sources without recompiling the program (assuming the same session).

As plugin modules they all have the same format and definition. The routine definition is:

```
short *get_block( short **num, int **new, int *runinfo );
```

The return value is a pointer to a short that contains upto 65536 elements. The value "new" is a variable to determine if a new file is to be read. The value "runinfo" is now depreciated.

If you, or a user, wants to follow the process by which these routines extract the data the debug toggle "GET" should be set to on. If additional information is required to be printed out the following formalism should be used.

```
if ( ( *debug_level & GET ) == GET )
 {
  printf( "get_block : About to open \"disc.info\" ...\n" );
 }
```

tape.so

As the name implies, this routine reads data from tape. The value "num" refers to the current tape number. Details about the files to be read from tape are obtained from the file `tape.info` which is stored in the tmp directory that is allocated to the current MIDAS session.

The return value from the "C" library function "read" (the number of bytes read) is used to determine the action. The segment of code that deals with this return value is shown below:

```
  switch ( bsize )
   {
    case -1:
     printf( "tape : Error, %s returned %i\n", tdev, bsize );
     exit ( 0 );
     break;
    case 0:
     if ( at_eof )
      {
        at_eov = 1;
      }
     at_eof = 1;
     mt.mt_op = MTFSF;
     mt.mt_count = 1;
     ioctl( fd, MTIOCTOP, &mt);
     break;
    case 80:
     if ( at_eof )
      {
       while ( num_tapes-- )
        {
         if ( ansi_fmt )
          {
           tnum = (short int)(runinfo[ ctr++ ]);
```

```
            if ( ( *debug_level & GET ) == GET )
             {
              printf( "tape : tnum = %i\n", tnum );
             }
            prun = &runinfo[ ctr++ ];
            memcpy( run, prun, sizeof( run ) );
            if ( ( *debug_level & GET ) == GET )
             {
              printf( "tape : Run = %s\n", run );
             }
            ctr++;
           }
         }
       }
      at_eof = 0;
      break;
     default:
      at_eov = 0;
      at_eof = 0;
      *new = (int *)tctr;
      *num = block;
      return block;
      break;
    }
```

As can be seen, it is assumed that all return values greater than 80 bytes refer to data. Obviously this is not completely true. For example some formats use upto 256 bytes as message blocks on tape. This should be born in mind when data decoders are being written.

disc.so

As expected this module reads data from a/multiple disk file(s). The information about the files to be read are stored in `disc.info` in the tmp directory of the current MIDAS session. Unlike the case of tapes, the data blocks in disk files can be accessed at any size the user tells the program. This can lead to the loss of data, so the information that is stored in `disc.info` *must* be acurate.

This routine also checks that the file that is to be accessed has a sensible size. If the file size is 0 then it will ignore that file.

midasTS.so

This module will read data from the shared memory allocated by the MIDAS TapeServer. It ascertains the relative age of the data so that if you are waiting for data to come into the TapeServer the same block of data will not be sorted over and over again. There could exist a case where you only sort a fraction of the data being sent to the TapeServer - this will only occur if the overheads for the sort are greater than the spead at which data is being passed to the TapeServer.

Routines to decipher the data

As with the routines that read the data the files described here are plugin modules. This being the case they all have the same definition:

```
short decipher_data( unsigned short *raw_data, unsigned int *id,
                     unsigned int *adc, unsigned int *numwrds )
```

raw_data is a pointer to the memory containing the data block. id is a pointer to an array containing the detector id data words. adc is a pointer to the array containing all of the data associated with each detect-

or id. nwrds is unused. The return value is the event multiplicity or equivalent.

Debug statements within this module type are defined using the following formalism.

```
if ( ( *debug_level & DECIPHER ) == DECIPHER )
 {
  printf( "decipher_data : Other data wrd\n" );
  printf( "              : id         = %i\n", id  [ 0 ] );
  printf( "              : Module no  = %i\n", adc [ 0 ] );
  printf( "              : Info Code  = %i\n", adc [ 1 ] );
 }
```

The following data formats are currently supported:

- Eurogam
- Exogam
- GREAT

Routines used during the sort

These routines can all be found in the Lib directory.

create_shared_memory	This routine creates the shared memory segments for the program. In order to allow more that one user at a time, the key comprises the tcl tmp directory that is created by the MIDAS-session and the letter 'a'.
create_spectra	This routine creates the named spectra via the memsas daemon. If the "overwrite" flag is set to 1 then any spectra that exist with the same names are deleted before being created.
end_prog	This routine will always be called when the program exits - even if there is an error condition. The routine ensures that we tidy up and print out any information that may be helpful to understand why the program crashed - if it did.
histogram_data	This routine will, if invoked, generate histogram spectra of the data being analysed.
parse_args	This routine handles the arguments that are passed to the program at the begining. It creates any control files that need to be and sets all of the appropriate flags.
plugin_loader	This is the routine that loads our shared libraries: the event decoder and the get_event routine. If any more shared libraries need to be loded than this routine should be modified.
prog_pid	This routine returns the process id for the program as it runs. This PID is required when the program is being controlled by signals (depreciated).
read_cfg_file	In order to histogram the data a configuration file, identical to that used by Exogam is required. This routine reads in such a file.
read_debug_file	This routine reads the debug file allowing the correct debug level to be set. The debug file is a text file containing a list of all of the possible debug options and a toggle flag (0/1).
read_specarray	The spectra that are to be created are extracted from the sort file and written to a seperate file. This file is then read when the sort is started. This file contains the filename, fileid, length and spectrum type for all

	of the spectra that we are intersted in creating.
sort-signals	This routine sets up the signal handlers for the sort, ensuring that all errors are caught.
sort_got_sig	This routine has now been depriciated.
swap	This file contains two routines that can be used to byteswap 16 bit and 32 bit words

Routines Available to the user

These routines can all be found in the Sorting directory.

gainmatch	The prototype for this routine is: *int gainmatch (float a, float b, float c, int val)*. i.e. if you provide the variables a, b, c and a value - val - to be modified the returned value will be related to the input val by: return = a + (b * val) + (c * val * val).
gate1d	This routine returns 0/1 id a data value (data) lies in a list of limits (limits). The routine has the prototype: *int gate1d(int *limits, int data)*. *limits is a 1D array consisting of 2n + 2 elements, where n is the number of lo-hi pairs, the routine lnows that the list is complete when it encouters two sequential zeros..
gate2d	Uses the routine pnpoly to determine if a vertex lies within a polygon. The prototype is *int gate2d(int *limits, int x, int y)*. Here x and y are the coordinates of the point to be tested, the array limits consists of 2n + 2 elements. Each element is (x,y) for the verticies of the polygon. The list should be terminated by two zeros.
inc1d	*void inc1d(int id, int val)*. This routine will increment chanel "val" in spectrum "id" by 1. The routine checks that the spectrum id and the channel are both valid.
inc2d	*void inc2d(int id, int xval, int yval)*. This routine is the same as inc1d but for a 2D spectrum.
incv1d	*void incv1d(int id, int chan, int val)* - increment channel "chan" in spectrum "id" by "val" counts.
incv2d	*void incv2d(int id, int xchan, int ychan, int val)* - as incv1d but for a 2D spectrum.
read2d	*int *read2d(char *fname, int set)* - Read in a data file containing a sequence of polygon vertices. The input file format is the same as that output by MIDAS. The data file is specified via the string "fname". If the file contains more than one polygon array, the value "set" specifies which one to use. The return is an integer array.
set1d	*void set1d(int id, int chan, int val)* - This routine will set the counts in channel "chan" of spectrum "id" to "val".
set2d	*void set2d(int id, int xchan, int ychan, int val)* - As for set1d but for a 2D spectrum.
val1d	*int val1d(int id, int chan)* - Returns the number of counts in channel "chan" of spectrum "id".
val2d	*int val2d(int id, int xchan, int ychan)* - As for val1d, but for a 2D spectrum.

Writing Sort Programs

Irrespective of the language that the sort program is written in (C or Fortran) it must have the following basic structure:

```
*trigger
128
*oned
1 mult 4096
2 adc  4096
4 val  16384
5 val  16384
*twod
3 tproj 4096 4096
*vars
*sort
int sortin( void )
{
}
```

Some of these "starwords" arise from a historic context and are currently not implemented: e.g. "*trigger" and "*vars". The "*oned" and "*twod" words are used to deliniate the definitions of 1D and 2D spectra. The "*sort" word is used to define the start of the users sort routine. Prior to "*sort" the file can be commented using the "#", after "*sort" comments must be written as if you were writing a "C" or "Fortran" program.

The user can specify routines called *init* and *finish* which are called at the start and end of the sort respectivly. If these two routines are not specified they are automatically added at compile time.

In addition any other routines that the user may wish to call can be specified in this file.

In general the best way to see how to use the different functions is to look at the example sort files. These can be found in the Examples directory.

In the following sections a detailed description of how to write sort programs in both "C" and "Fortran" is presented.

Global Variables

In order to make the data that has been read from Tape/Disk/Memory available to the users sort routine three "global" variables have been implemented. The method by which these are accessed is determined by the programming language that you are using. These variables are:

gid	This is an integer array containing a list of the ADC id's. Note that this is often the "raw" id and would need further decompisition to obtain numbers that the user would be more familier with.
gdata	This is an integer array containing all of the data words that are associated with a particular detector id. A maximum of 1024 wrds per id is allowed.
mult	This is the number of gid - gdata word pairs.

Using "C"

A sample sort file written in "C" is shown below:

```
#--> ================================================ <--#
#--> Sample sort program, written to extract pulse data  <--#
#--> from the data taken in Koln for the GRT test !      <--#
#--> ================================================ <--#
*trigger
#--> ================================================ <--#
#--> this *word is not currently implemented             <--#
#--> ================================================ <--#
128
*oned
#--> ================================================ <--#
#--> Here we shall define our 1D spectra                 <--#
#-->                                                     <--#
#--> This can be achived in a couple of different ways   <--#
#-->                                                     <--#
#--> id name len         - Make sure the ID's are unique <--#
#-->                                                     <--#
#--> >> A list of spectra called grpnameidstart ....     <--#
#--> idstart...idstop grpname len                        <--#
#-->                                                     <--#
#--> ================================================ <--#
1 mult 4096
2 adc  4096
4 val  16384
5 val  16384
*twod
#--> ================================================ <--#
#--> 2D spectra are defined in the same way, using the   <--#
#--> *twod word. You can specify the size of the x and   <--#
#--> the y axis.                                         <--#
#--> ================================================ <--#
3 tproj 4096 4096
*vars
#--> ================================================ <--#
#--> This *word has been depreciated                     <--#
#--> ================================================ <--#
*sort
//--> ================================================ <--//
//--> So this is where the sort is defined, note the     <--//
//--> change in our comments .....                       <--//
//-->                                                    <--//
//--> -- First include any header files that are reqd    <--//
//--> ================================================ <--//
#include <stdio.h>
#include <stdlib.h>
//--> ================================================ <--//
//--> So we can define the init routine, all we shall    <--//
//--> do is to print out an inane comment to let us      <--//
//--> know where we are!                                 <--//
//--> ================================================ <--//
int init(void)
{
 printf( "In user init ...\n" );
    return 0;
}
//--> ================================================ <--//
//--> Now define the sortin routine. Remember this is a  <--//
//--> must, if it is not included your sort will not     <--//
//--> work!!                                             <--//
//--> ================================================ <--//
int sortin(void)
{
 //--> ================================================ <--//
```

```
//--> Define some variables for use within this         <--//
//--> routine. - Only declare what you need!!           <--//
//--> ================================================ <--//
FILE *f;
int t = 0;
int i, j;
static int ctr = 0;
char   fname[ 128 ];
//--> ================================================ <--//
//--> Lets increment a 1D spectrum with our event mult  <--//
//--> ================================================ <--//
 if ( mult > 0 && mult < 4096 )
   {
    inc1d( 1, mult );
   }
//--> ================================================ <--//
//--> Lets increment a 1D spectrum with the id's of the <--//
//--> ADC's extracted from the event.                   <--//
//--> ================================================ <--//
 for ( i = 0; i < mult; i++ )
   {
    if ( gid[ i ] >= 0 && gid[ i ] < 4096 )
      {
       inc1d( 2, gid[ i ] );
      }
   }
 if ( *nwrds  > 0 && *nwrds  < 16384 )
  {
   inc1d( 5, *nwrds  );
  }


 for ( j = 0; j < mult; j++ )
  {
   for ( i = 5; i < 500; i++ )
    {
     if ( gdata[ i + ( j * 1024 ) ] -7800 > 0 &&
          gdata[ i + ( j * 1024 ) ] -7800 < 4096 )
      {
       inc1d( 4, gdata[ i + ( j * 1024 ) ] -7800 );
       //--> ========================================== <--//
      //--> Now increment a 2D spectrum ...               <--//
      //--> ========================================== <--//
        inc2d( 3, i,  gdata[ i + ( j * 1024 ) ] -7800 );
      }
     if ( ( ( gdata[ i + ( j * 1024 ) ] - 7800 ) > 1000 ) &&
          ( ( gdata[ 10 + ( j * 1024 ) ] - 7800 ) < 100 ) )
      {
         t = 1;
      }
    }
   if ( t )
    {
     sprintf( fname, "/home/dea/pulse%i.txt", ctr++ );
     f = fopen( fname, "w" );
     for ( i = 5; i < 500; i++ )
      {
       fprintf( f, "%4i %i\n", i, gdata[ i + ( j * 1024 ) ] );
      }
     fclose( f );
     if ( ctr > 10 )
      exit( 0 );
    }
  }
```

```
 return 0;
}
//-->================================================== <--//
//--> Define the finish routine                        <--//
//-->================================================== <--//
int finish(void)
{
//-->================================================== <--//
//--> Just print something out to let us know ...      <--//
//-->================================================== <--//
  printf(" Time to gooooooooooo\n" );
    return 0;
}
```

Using "Fortran"

ToDo List

i. Complete this manual.

ii. Find a permanent maintainer for this package.

iii. Increase the list of decipher routines.

iv. Increase the list of user libraries.

v. Add support for writing data back out.

vi. Check migration to windows.