Using VSIB Test Software

Ari Mujunen, amn@kurp.hut.fi
Metsähovi Radio Observatory

02-Feb-2006

Abstract

This document describes the steps required to use the Debian/GNU/Linux 3.0 based MVR recording computer to test VSI-H DOM modules using VSIB data acquisition board test programs.

Contents

1 Introduction

Please find below instructions how to use a MVR ("Metsähovi VLBI Recorder") Debian/GNU/Linux 3.0 computer to capture and examine data from VSI-H compliant DOM modules.

2 Setting Up

The computer (originally a VSIB board test PC called "commodus.kurp.hut.fi" at Metsähovi) is an Athlon64 machine with a Epox 8KDA3J nVidia nForce3 250 GB based motherboard, 1GB of DDR memory (single-channel) and a 40GB PATA Maxtor system disk plus three 200GB PATA Maxtor data disks. All disks use generic plastic PATA swap enclosures, and the empty /dev/hda slot has one extra empty enclosure to enable special disk tests and disk copying. The system disk is connected via an Abit PATA-to-SATA converter into the motherboard /dev/hde native nForce3 SATA connector.

The system is set up for automatic network configuration via a DHCP server, so after connecting 230VAC power cable, PS2 (by default Italian) keyboard, PS2 wheel mouse, any text-mode capable VGA monitor or panel, and a 10/100/1000 Mbps Ethernet cable it should boot automatically into regular Linux virtual console text mode, with sshd active for remote connections.

The system has two predefined user accounts, 'root' and 'amn', the passwords of which have been written onto a taped Post-It note on the PC front panel. Regularly you should login as a regular non-root user; the predefined account supplied for this purpose is called 'amn', to keep in sync with the development environment at Metsähovi.

Should you need a static IP setup instead of automatic DHCP, you can log in as 'root' and edit the files '/etc/network/interfaces' to contain the correct IP numbers. The required lines are present in the file and they have just been commented out with '#' characters. DHCP additionally automatically updates DNS name server information in '/etc/resolv.conf', so if you change '/etc/network/interfaces' from DHCP into static IP, you should also add the IP addresses of your name servers into '/etc/resolv.conf'.

The PC has one VSIB board preinstalled, with the primary input VSI-H connector available in the regular PCI expansion board back panel slots, and the auxiliary (output/chaining) VSI-H connector (inside, on the inside edge of the VSIB board) being routed out with a short VSI cable, through a special cut-out slot.

Another special back panel slot cover with four LEDs and one USB connector has been provided. This helps in debugging by making the VSIB board stopped-waiting-running "traffic lights" visible without opening the PC cover. The USB connector has copies of VSI-H input clock and FIFO full debug signal available for scoping. These can be used by taking a regular USB cable and cutting and peeling the wires visible for scope probes. A sticker near the connector documents the wire colors vs. signals. (Green=clock, white=FIFO full.)

After boot, log in as 'amn' and proceed to directory 'cd proj/vsib'. You can make use of multiple virtual consoles by switching between them with Alt-F1, Alt-F2, Alt-F3 etc., up to Alt-F6, Alt-F1 being the default after boot. You can get rid of the decorative graphic top part of the screen with the command 'reset'.

3 Recording Test Data

The directory 'proj/vsib' contains numerous files (pardon for that!) but it is a snapshot copy of the VSIB software development directory we have here. The VSIB is

controlled directly by just two C source code files, 'vsib.c' which is a character-mode Linux device driver and 'wr.c' which is the record/playback user mode program. A third file 'vsib' (actually a device special node) has been provided also in the /home/amn/proj/vsib/ directory:

```
commodus$ ls -l vsib
crw-rw-r-- 1 root kurp 254, 0 2001-11-02 12:13 vsib
```

This device file name is used with user mode programs as standard input (or output), the device name is not hard-coded in those, e.g. './wr':

```
./wr ... < vsib    (read from VSIB, write on disk)
./rd ... > vsib    (write to VSIB, read from disk)
```

The compiled driver 'vsib.o' has been loaded automatically by one-line invocation in '/etc/init.d/hwtools' of the script '/root/vsib' which takes care of both loading the driver and mounting the three-disk software raid0 array:

```
#!/bin/sh
# Buff for 2sec 18MHz 32-track playback.
insmod /home/amn/proj/vsib/vsib.o bigbufsize=144000000
# echo "" > /proc/sys/vm/bdflush
mount /dev/md0 /i1
```

You can modify this '/root/vsib' as 'root' user should the need arise, but for regular testing this should be just fine as-is.

3.1 Connecting the VSI-H Device under Test

If you take a VSI-H cable and connect a VSI-H DOM device which outputs VSI bit stream data together with VSI clock into the back panel primary VSI-H connector, you will see that the data and clock are being immediately duplicated and repeated at the auxiliary VSI connector, appearing at the short VSI extension cable protruding from the PC back panel. This is provided primarily for chaining multiple VSIB boards to capture one VSI-H connector signal but it can certainly be used for debugging, to scope how the VSIB board input "sees" the signals presented in the primary back panel VSI-H connector.

The PVALID VSI signal in input connector must be asserted for the VSIB to clock in data. This feature is mainly used to drop unneeded data such as Mark4 formatted parity bits, but as a consequence if the VSI-H device does not assert PVALID, no data will be recorded.

The VSI clock can be in the range 0..64 MHz (and probably even slightly higher) and it doesn't have to be symmetric or continuous. Data is clocked in at clock rising edges.

There is a special debug mode available on the VSIB board: if you connect pin 1 of the JP2 "TEST" pin header connector at the upper part of the board to GND (available as pin header JP3, labeled with "GND" on-board) with a clip wire, the auxiliary VSI-H connector starts emitting the VSI TVG test pattern, generated at the rate of the 50 MHz internal quartz oscillator and with a synthetic 1pps pulse generated by a free-running downcounter, counting from 32000000-1 to zero, emulating the behavior of VSI 1pps at 32 MHz sampling clock, i.e. the 1pps pulse occurs every 32000000 VSI words. In this way you can get TVG data out of the

auxiliary connector (short VSI cable coming out of the PC), instead of the copy of input which is regularly there. The data output (be it TVG data or duplicated data from the primary input) of the auxiliary VSI-H connector is unaffected by VSIB board operation, that is, regardless whether the VSIB board is stopped, waiting, or running. Please note that the TVG VSI data stream being outputted in this case is indeed "overclocked" at 50 MHz.

3.2 VSIB Board Operation

The VSIB board has a very simple operating model, one settable 32-bit word of settings and an internal state, one of three possible states: stopped, waiting, running.

Regularly, the board is in the "stopped" state where it resets all its internal logic to a known state. This is reflected in the back panel LED number 1 (and also on-board D4). At any time the driver software can reset VSIB back to this state by just writing into the 32-bit settings/mode word.

The board is started by writing another bit combination (the start bit together with all the settings required) into the 32-bit settings/mode word. The on-board logic lights up the panel LED number 2 (and also on-board D5) and starts "hunting" for the next occurrence of VSI 1pps pulse, keeping the logic in this "waiting" state.

The first data word being clocked in with the VSI 1pps signal asserted is the first word being captured and this triggers the board into the "running" state and lights up the panel LED number 3 (and also on-board D6). The board starts pushing incoming VSI words into its on-board 4 kB FIFO and from there into the PCI bus using DMA. The words land into a Linux "bigphysarea" main memory buffer where the 'vsib.o' driver finds them and presents them to the user mode programs via regular 'read()' calls. The main memory buffer is organized as a "scatter-gather" circular buffer and it is allocated when the 'vsib.o' device driver is loaded with the argument 'bigbufsize':

```
insmod /home/amn/proj/vsib/vsib.o bigbufsize=144000000
```

The above allocates 144000000 bytes for the ring buffer; generally at 512 Mbps less than 10 Mbytes is sufficient. The maximum consumption of this is being tracked with kernel log messages which appear in Linux virtual console number 1 (Alt-F1) and are also recorded in /var/log/kern.log:

```
vsib: big secondary ring buffer filled to 3456000 bytes
```

Should this reach a value very close to 144000000 it very probably indicates that the disk subsystem is not keeping up with the VSIB PCI DMA. A set of three Maxtor 200GB PATA disks can sustain a little over 700 Mbps at the beginning of disks but at the inner tracks, during the last 2% of disk capacity the sustained performance is very close to 512 Mbps. Another typical reason for sudden disk slowness is the attempt to overwrite large old files with new files with the same file names: deleting large files "on-the-fly" takes a relatively long time and the disk write process is left behind the PCI DMA rate.

The panel LED number 4 (and also on-board D7) indicates that the on-board PCI FIFO has overflowed. This could happen if the PCI bus is unable to DMA data quickly enough from VSIB into main memory. nVidia nForce chipsets are quite good at this, though, and this problem starts to appear at closer to 800 Mbps.

The 32 VSI-H bit streams are mapped into Intel little-endian format in such a fashion that a 32-bit integer (31/msb..0/lsb) in VSI-H can be read and handled like a 32-bit C integer in user mode code. In practice this means that VSI bit numbers 31..0 land in four consecutive bytes of physical memory as 7,6,5,4,3,2,1,0 / 15,14,13,12,11,10,9,8 / 23,22,21,20,19,18,17,16 / 31,30,29,28,27,26,25,24 and that they can be masked with four-byte hexadecimal integer masks and shifted and rotated freely in C code.

In the 32-bit settings/mode word of VSIB board there are three individually settable bits, one four-bit mode selector, and a 16-bit "skip counter". The './wr' recording program provides a rudimentary command-line interface to set each of these separately. That is, './wr' takes its command-line arguments and forms the 32-bit settings/mode word it uses to both setup and start the board according to the mode desired. These are described below together with the description of the './wr' command line.

3.3 The '/i1' raid0 Data Array Area

The three PATA 200GB Maxtor disks (/dev/hdb, /dev/hdc, /dev/hdd) have been partitioned with equal-sized primary partitions /dev/hd{bcd}1, which in turn have been combined with the help of Linux software raid0.o and the 'mdadm' tool to provide 3x200=600GB of interleaved raid0 disk space under the device name /dev/md0. This has been formatted with the regular 'mke2fs -m0 /dev/md0' Linux filesystem and it is being auto-mounted for use by the /root/vsib script containing a regular 'mount /dev/md0 /i1' command.

By convention and as 'root', one test subdirectory has been created inside /i1, '/i1/t', and this directory has been made 'chown amn.kurp /i1/t', thus it is possible to create and delete test data files in this directory as a regular user 'amn', without the fear that one might mess up with system files while working as 'root'.

The directory contains a few pre-recorded VSI TVG test pattern files which may be useful for comparisons.

3.4 Invoking './wr' Recording Command

While in directory 'proj/vsib' (/home/amn/proj/vsib) if you just invoke the './wr' command as:

```
./wr
```

you will get a quick refresher of the required command line arguments:

```
./wr: needs at least blocksize totalblocks mode skip embed giga ndirs { path%d blks }
```

The following is a description of each of these arguments.

blocksize The number of bytes being used to read() from vsib.o device driver and to write() to disk files. Performance is largely unaffected regardless whether this is large or small. Typically it is convenient to use something with a relation to the data being recorded, e.g. its block size. For instance, because

Mark4 data has 20000 consecutive bits in one frame (without parity), a typical value often used as block size is 80000 bytes, since 20000 32-bit VSI words (comprising of one frame of 32-track data) would need 80000 bytes. This will also be divisible with 32000000 words per second, that is, the 32 MHz VSI clock.

totalblocks This is the total number of "blocksize"-sized blocks to be recorded. After writing this many blocks to disk, './wr' stops the VSIB board and terminates.

mode The 32/16/8/4/2/1-bit mode (bit mapping) to be used. This is the 4-bit "mode" field found in VSIB board 32-bit settings/mode word. Please see the section 3.5 about VSIB mode bits on the following page.

skip After clocking in one VSI word (32/16/8/4/2/1-bit), skip this many word clocks before accepting the next VSI word. Can be in the range 0..65535. This is the 16-bit "skip counter" found in VSIB board 32-bit settings/mode word. This is used to "decimate" the input clock. For instance, if the channel bandwidth is set to e.g. 4 MHz and it is still sampled at 32 MHz VSI clock, "skip=3" can be used to divide the effective sampling clock by 4, resulting in an effective 8 MHz VSI clock.

embed Setting this to '1' causes the VSIB to replace each VSI word clocked in while the VSI-H 1pps signal was asserted with a word with all-ones. (The number of 1-bits actually recorded depends on the "mode" setting, i.e. in e.g. 16-bit mode a word of 16 one bits appears in the resulting data.) This "1pps marker" can be used to detect slipped and inserted VSI words (which usually originate in damaged VSI clock signal, e.g. missed or extra clock pulses due to e.g. exceeding the LVDS signaling common mode voltage range). Please see the description of 'ck1pps.c' program in section 4.4 on page 10.

giga Setting this to '1' alters the way VSIB echoes the incoming VSI signals in the output auxiliary VSI-H connector. In the "giga" mode the data bits 31..0 are delayed by one clock with respect to the 1pps signal. Thus a second VSIB chained after the first one with the "giga" bit on starts recording one VSIB word late. When this is combined with "skip=1", the net effect is that the first VSIB PC will record even VSI words and the second VSIB PC will record odd VSI words. The chain can be made arbitrarily long by just increasing the "skip" value to the number of chained PCs. With "giga=1", "skip=3", and four VSIB PCs a 64 MHz 32-bit double-speed 2 Gbps VSI-H signal can be easily recorded.

ndirs This is the number of locations / path names './wr' where will record the blocks in round-robin fashion. This effectively emulates the natural behavior of raid0 which writes consecutive blocks in all the disks in raid0 in round-robin fashion. Since it makes little sense to do this "interleaving" twice, together with raid0 this is usually set to '1'.

path blks For each "ndir" set in the above, a pair of pathname and number of blocks to be recorded in this pathname must be set. The "path" path name string here can (and usually should) be embedded with a '%d'-style integer sprintf() format directive. When "blks" number of blocks has been written to a single file it will be closed and another file will be opened, using the "path" string as a file name template. The first file gets the '%d' part replaced with zero, the next one with one, and so on. A typical example would be '/i1/t/test-%05d' which will result in file names like '/i1/t/test-00000', '/i1/t/test-00001', '/i1/t/test-00002',...

Typically you would invoke './wr' as follows:

```
./wr 80000 10000 2 0 0 0 1 /i1/t/test-%05d 5000 < vsib
```

This would record the 16 least significant bits of VSI at full clock rate into two files, /i1/t/test-00000 and /i1/t/test-00001, 5000*80000=400000000 bytes each. The total duration of this recording would be 12.5 seconds. Please note that './wr' expects the name of the VSIB device file as its standard input, otherwise it will complain "standard I/O is not an VSIB board".

3.5 VSIB Board Modes

The VSIB 4-bit "mode" register is documented in 'proj/vsib/docs/modes.txt' and the following modes have been defined in VSIB/R recording Xilinx firmware.

```
all32bits:      "0000"; -- MK5 32-bit mode, all the VSI data
even16bits:     "0001"; -- MK5 16-bit mode or all sign bits from VLBA sampler
low16bits:      "0010"; -- alt: 2-bit USB from 8 BBCs
                        -- norm: 2-bit USB+LSB from 4 BBCs
                        -- also S2 16-bitstream mode
low8bits:       "0011"; -- alt: 2-bit USB from 4 BBC:s
                        -- norm: 2-bit USB+LSB from 2 BBCs
loweven8bits:   "0100"; -- MK5 8-bit mode or 8 sign bits from VLBA sampler
                        -- alt: 1-bit USB from 8 BBCs
                        -- norm: 1-bit USB+LSB from 4 BBCs
autoc4x2bits:   "0101"; -- as 'low8bits'; Autocorrelator 2-bit mode
autoc4x1bit:    "0110"; -- 4-bit low even; Autocorrelator 1-bit mode
test2bits:      "0111"; -- 2-bit mode for test
test1bit:       "1000"; -- 1-bit mode for test
VLBA_142:       "1001"; -- VLBA 1:4 fanout mode with 2-bit sampling
VLBA_141:       "1010"; -- VLBA 1:4 fanout mode with 1-bit sampling
VLBA_122:       "1011"; -- VLBA 1:2 fanout mode with 2-bit sampling
VLBA_121:       "1100"; -- VLBA 1:2 fanout mode with 1-bit sampling
VLBA_112:       "1101"; -- VLBA 1:1 fanout mode with 2-bit sampling
```

Of these, the modes 0 ("all32bits") and 2 ("low16bits") are probably the most useful for testing VSI-H DOMs.

When you try to use the mode 0 to capture all 32 bit streams of VSI you will have to use "skip=1" to halve the VSI clock to 16 MHz and capture effectively every other word when using just one VSIB PC. Although VSIB will manage up to 800 Mbps in a nForce3 PC, the PCI32/33 bus just cannot sustain full 1024 Mbps and the back panel LED number 4 (PCI FIFO overflow) will light up.

3.6 Delayed / Timed Start

It is often desirable to be able to start VSIB capturing at a given UTC time. For this you need to ensure that the PC gets its time from a "good enough" (within 0.1 second) NTP time server. You should change ntpd configuration to point to your local NTP time server. It is easiest to do this (as root) with 'dpkg-reconfigure ntp-simple' which will update both '/etc/ntp.conf' and '/etc/default/ntp-servers' consistently. After updating the files you can '/etc/init.d/ntp stop', '/etc/init.d/ntpdate start', '/etc/init.d/ntp start' and wait for about 5–10 minutes for the ntpd to gain sync, but it is probably easiest to just reboot with ctrl-alt-del. Oops, I have forgotten the '/etc/init.d/ntpdate' script in disabled state, it is now called '/etc/init.d/ntpdate.not-in-csc', so please rename/'mv' it back to '/etc/init.d/ntpdate'. ('ntpdate' gets the

initial clock value from your NTP server during PC reboot so that the regular ntpd syncs up faster.)

Once your PC has a sufficiently (within 0.1–0.2 seconds) correct UTC time you can use the './dstart' command to delay the startup of './wr' in the following fashion:

```
./dstart 2006-02-05T23:00:00; ./wr ...
```

'./dstart' sleeps until 0.5 seconds before the specified ISO 8601 UTC date/time and then exits, allowing the next command to start just before the next 1pps is to be expected. (It actually sleeps for half of that time and then re-checks the computer/NTP clock and sleeps again for half of the remaining time, in loop, thus keeping you informed about the time left.)

You can take a look at the './recexpt' script provided as a sample how to record a complete "schedule". It basically lists passes like this:

```
./dstart 2004-04-19T18:45:00; ./recpass No0113-${HEADSTK} 220
./dstart 2004-04-19T19:00:00; ./recpass No0114-${HEADSTK} 220
./dstart 2004-04-19T19:15:00; ./recpass No0115-${HEADSTK} 220
...
```

'./dstart' is used to synchronize invocations of the './recpass' script to UTC, and './recpass' in its turn predefines the desired './wr' arguments.

There are additional './ndate' and './ndatesec' programs which print out the current computer UTC time in ISO 8601 format, './ndatesec' with fractional seconds ("2006-02-02T08:59:34.919841") and './ndate' rounded up to the next second change ("2006-02-02T08:59:35"). You can use these for instance to timestamp recorded files, e.g.:

```
./dstart ...; ./wr ... /i1/t/test-`./ndate` ...
```

4 Inspecting Captured Data

Since all the data is captured in regular Linux files we can use all the available Linux tools to inspect the files. Furthermore, there are several small utility C programs in the 'proj/vsib' directory which make VSI-specific debugging easier.

4.1 Displaying and Dumping File Contents

You can use the regular 'od' and 'hexdump' commands to get Intel little-endian dumps of the data files. 'hexdump' gives more "typical" hexadecimal results than 'od -x' and it is more flexible (adjustable formatting, '-s' skip bytes in the beginning etc, please see 'man hexdump'). You will typically use these like:

```
hexdump /i1/t/test-00000 | less
```

to get paged output. A neat trick is to dump to temporary text files and compare the result with 'diff' to locate where the differences are:

```
hexdump /i1/t/test-00000 >/tmp/t0
hexdump /i1/t/test-00001 >/tmp/t1
diff /tmp/t0 /tmp/t1 | less
```

A major drawback in interpreting these hexdumps is that the Intel little-endian format is not being displayed in "straight row" but instead bytes 3210 re-ordered as 0123. Thus I have written "bit-mode" dump programs d32.c, d16.c, d8.c. These output the standard input interpreted as 32/16/8-bit integers and dumped bit-wise, one word on each line.

```
./d16 < /i1/t/test-00000 | less
```

The MSB is dumped first (on the left), the LSB last (on the right). To search for a specific bit pattern in 'less' you can type '/' followed by the '0' and '1' bit characters, each separated by a space. Quickly scrolling the output by repeatedly pressing "space" will quickly reveal VSI bits stuck to one or zero, since they will stay on/off "in column" in the constantly refreshing screen output. Should you need to know the VSI word number of each word you can line-number the output with:

```
./d16 < /i1/t/test-00000 | cat -n | less
```

4.2 Comparing Files

In addition to 'diff'ing the hexdumps (described above) you can directly compare binary files with 'cmp'; by default it will report the first byte location where the files differ, please see 'man cmp' for further options.

To quickly ensure that files of a repeated test (such as repeatedly capturing TVG test pattern) are indeed the same you can use 'md5sum' on a set of files and verify that you get the same checksum on all files. Should 'md5sum' refuse to open a file which is larger than 2GB you can pipe it with 'cat /i1/t/test-00000 | md5sum'.

4.3 Word Slips and Insertions

If you can set your VSI-H DOM to output a free-running 32-bit counter as its data bit outputs there are special programs 'dd32.c' and 'dd16.c' to help in detecting missing and inserted VSI words. (These are most probably caused by problems in VSI clock, thus it makes sense to connect a digital scope to the VSIclk debug output available via the special back panel USB connector and set the scope to hunt for clock pulse duration errors.)

'./dd32' will interpret its standard input as 32-bit unsigned integers and dump for each word a line consisting of the word's decimal offset (in words, first word numbered as zero), the word itself and the next word in hexadecimal, and finally the difference between the word values in decimal, "first - second". With a free-running counter the difference should always be the same, e.g. 00000001 when "skip=0" and 00000002 when "skip=1". ("skip=0" and 32 bits is not possible unless VSI clock is 16 MHz or less.) You can use the following to find only those locations where the difference is not the expected one:

```
./dd32 < /i1/t/test-00000 | fgrep -v 00000002 | less
```

You might get lines like:

```
20: 4C24202A - 203A676F = 736737467
```

from where you can easily see that the problem location is the 21. word and the offending word values are the hexadecimal ones shown.

4.4 Checking for Embedded 1pps Markers

Another check for longer data runs is to check for the correct occurrence of the embedded 1pps marker enabled by "embed=1" while recording with './wr'. You can use the utility 'ck1pps.c' which can scan even longer files reasonably quickly for deviations of the occurrence of 1pps marker words. It is used as follows (oops, I seem to have forgotten to compile it, you can do it with 'make ck1pps'):

```
didius:~/proj/vsib> ./ck1pps
./ck1pps: needs at least firstseek blocksize totalblocks wordbits
didius:~/proj/vsib> ./ck1pps 0 64000000 300 32 < /i1/t/test-0000
```

firstseek is the number of bytes to skip in the beginning of the file, usually 0.

blocksize is the number of bytes in the recording block of 1 sec of data. 64 million bytes corresponds to 16-bit data at "skip=0" or 32-bit data at "skip=1" with 32 MHz VSI clock.

totalblocks How many blocks to scan, in terms of "blocksize". While in sync, './ck1pps' only reads "wordbits" amount of data at the start of each block so it will execute relatively quickly.

wordbits Can be 64/56/48/40/32/24/16/8 bits per word.

5 Further Pointers

Please take a look at '/home/amn/proj/docs' subdirectory where you can find a few additional documents:

pre-sw-inst.pdf Lengthy instructions as how to install Debian 3.0 with CD-ROMs; somewhat outdated, originally written for Paul Burgess (Jodrell).

vsib-broch.pdf VSIB board brochure.

modes.txt VSIB and VSIC board modes (definitions of those four mode bits).

vsib-back-panel-display.txt How to modify a MSI motherboard diagnostic LED back panel into a VSIB debug LED panel.

In the main '/home/amn/proj/vsib' subdirectory you can also find a few additional programs which might be useful:

File2net You can use this to stuff data files into Mark5A in 'disk2net=' mode. (This is 2003-03-18 version of the Haystack code so a newer version copied from Mark5A source code might be a good idea. Nevertheless, it seemed to compile ('make File2net') without modifications.)

cxextr.c Extract two 2-bit sample streams from n-bit files, output two 4-bit samples (one byte) per each input word.

exb.c Extract bits from files, output bytes per 1/2-bit samples, byte values can be defined on command line. Might be useful to export 1/2-bit sample streams to external tools like Matlab for further (e.g. spectral) analysis.

fixmk5crc{8,16,32,64} Fix 8/16/32/64 track wide Mark5A format files to contain zero rack number in header and recalculate CRC-12 checksums in every frame header. Can be used to recover recordings which cannot be played back with Mark5A because the rack number is not even. Serves also as a model for a short program to manipulate longitudinal bits of VSIB recorded files.

ft{1,2,3}.c Fourier transform programs to get autocorrelation spectra out of bit sampled data. Mainly interesting is ft3.c which was the basis for Beppe's 'chkchk' program, I think. It can extract 2-bit data and integrate autocorrelation spectrum out of that.

mrg.c Takes several files (originally recorded with a chain of VSIB PCs with "skip=" and "giga") and merges the files back into one.

sf.c Together with sff.c, forms the Mark4 software formatter which was used in selected 512 Mbps Mark4 formatter modes during the Huygens descent to Titan experiment. Can be used to format raw sampled data into Mark4 format, ready for stuffing into a Mark5A via File2net for subsequent playback with Mark5A.

skip.c Shows how to externally convey information into running process of './wr'; has been used to reposition the playback point within files during playback './rd' and it uses POSIX shared memory to achieve the IPC communication.

trunc.c A simple program to truncate away the tail part of any given file (even if larger than 2GB). Simply a wrapper around the 'trunc64()' library call.

To find out more about these programs it is really best to see the source code; all of these are quite short, a few hundred lines of C at most.