TELEMETRY USING SOFT REAL–TIME

TELEMETRY BASED ON SOFT REAL–TIME DEVICE DRIVER

Dipl. Ing. Jürgen Keidel
EADS Deutschland GmbH
Military Aircraft Business Unit
MT5 Flight Test

wizant.keidel@01019freenet.de
juergen.keidel@m.dasa.de
www.01019freenet.de/wizantkeidel

ABSTRACT

Telemetry requires only soft realtime, i.e. receiving data at a fixed high rate without loss . These data must be processed in time and delivered to several different applica–tions. The requirement for being real time is weak, as seen by human eye. Therefor standard LINUX is used with a special driver. The driver uses a SBS–4422 decom–mutation board and delivers data blocks in real time into user–space using cyclic buffer and DMA. The associated application is forced into RT–scheduling (SCHED_FIFO) and uses a standard read to synchronize and to get the buffer pointer. Measurements show that at actual data rates the rt–jitters stays in the range of 1 ms, with some exceptions, which are covered by the ring buffer. At the actual data rates (512 words every 2 ms) no data are lost, even if the remaining system is heavily loaded.

INTRODUCTION

Telemetry has a wide area of applications. Within this document, telemetry is done for flight test, so transferring data rates up to 4 Mbit per second as PCM data stream. Herein the data are packed into so called frames of fixed size, each one identified by a sync–pattern and a subframe id. Within the frame the data are transferred with a fixed word size. So at first, the incoming bits have to be synchronized and packed into a frame, then the data words need a lot of processing as calibration, combining and splitting. The last part delivers the final data to all the applica–tions, which will display them. In fact, delivering data deals with more than tenthousand different measurement data.

The first part, synchronizing the pcm–stream is done by some special hardware. The second and most time consuming part was done in former times by special hard–ware, but now it is a job for software as well as the third part. At the point, where the hardware has constructed a frame and delivers it to the application, realtime is requested as the data packages won't ever wait for a user. But calibration forces this job into user space, as it needs floating point operations.

PREVIOUS TELEMETRY SYSTEM

The system, being used since years was based on special hardware from Aydin Monitor, a Silicon Graphics server, reflec–tive memory from Vmic and several workstations from Silicon Graphics.

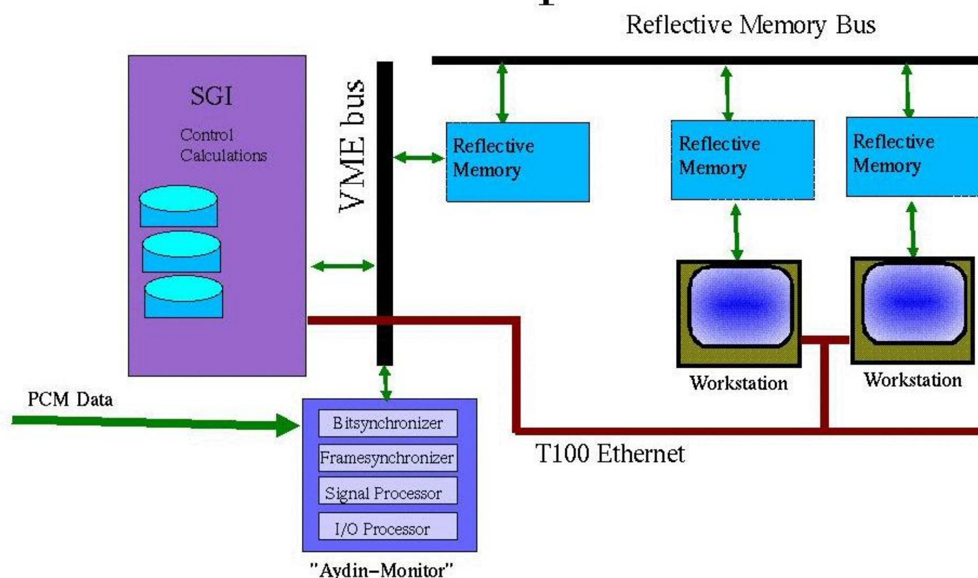
Going into detail: The incoming PCM–data stream, a serial bitstream, is prefor–matted in the first module of the "Front–end", the bitsynchronizer and fed into the decommutator which synchronizes the bit stream and breaks it down to "frames", feeding now the single words, which are completed with a unique identifier into the "Signal–processor", a fast bit–sliced proc–essor which calibrates, combines and splits the data into their final format. Finally these data reach the i/o–processor, which pushes them either into reflective memory or into normal memory for disc output.

PROBLEMS

Now, since the system is more than nine years old, several problems arise. First of all The "Front end"; it is impossible to get spare parts or to upgrade it. Second the explosion in computer speed results in a similar explosion in data rates. And third the costs. The actual system, just calcu–lating the "front end" and the server, sums up to something like 250 000 $ and up.
So what could be done?

There is standard hardware on the market fulfilling some of the tasks, the actual "Front end" is doing.
Modern PC's are powerful and fast enough to serve as Signal–processor as well as as data server.

This hardware would be much cheaper and all the basic software, application and control software would be still usable in most cases "as is".



SOLUTION

Choosing LINUX as operating system resulted in the possibility to write the necessary drivers ourselves and, as it is a "unix", continue with all the existing software. During that time, there was no discussion about various RT–versions of LINUX, but the standard linux was at least a good start.

LINUX BASED TELEMETRY SYSTEM

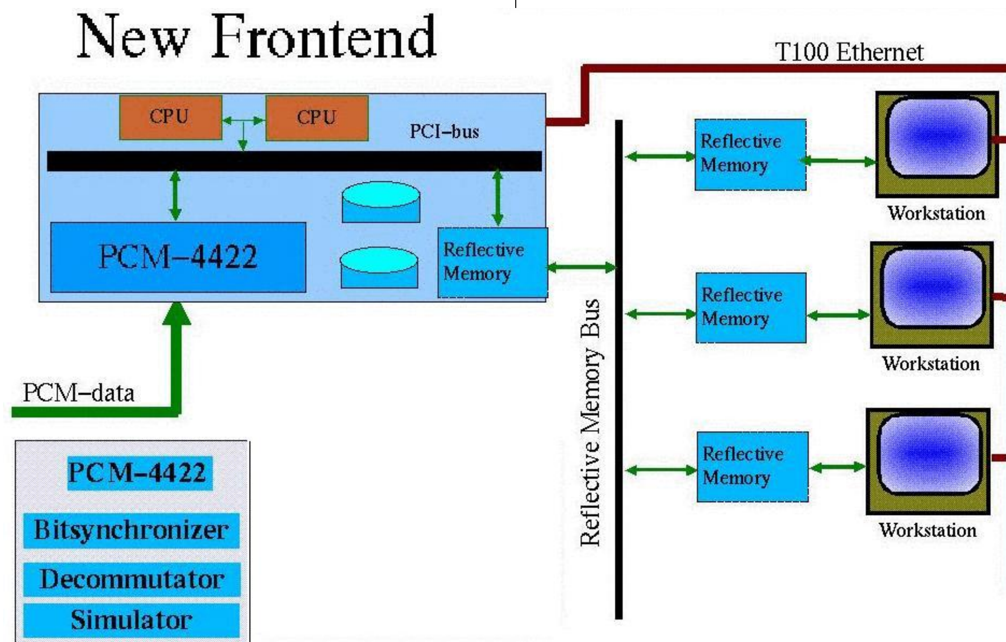
.

3com T100 network card
adaptec SCSI

All parts are standard products, and may be replaced at any time be new, better or faster ones. The amount of money is about 40 000 $ max for the front–end and server side.

SOFTWARE

Actual used is SusE Linux 6.4 with kernel version 2.2.14.
No changes to the operating System
Only Drivers are needed, written as loadable modules.

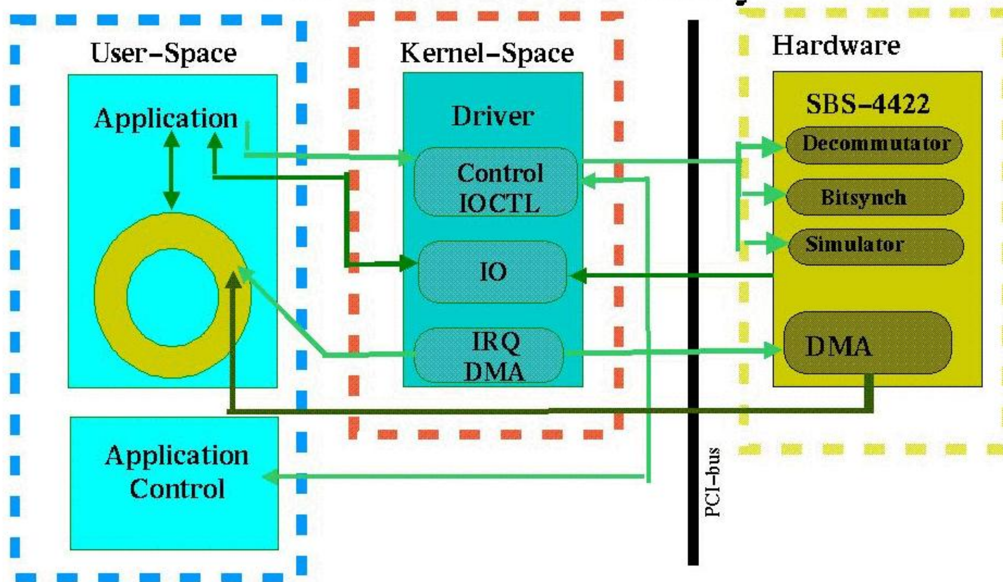


Hardware

Industrial dual pentium 600 MHZ
128 MB ram
SBS_4422_pci decommutator/bitsync/...
Vmic reflective memory

All applications could be used "as is" after recompiling.
Only the control programs for the new hardware had to be developed.

Driver definitions



The driver has several different regions of functionality.

1) control of hardware (done via IOCTL)
2) I/O (READ)

3) RT–control to guarantee the data rate.

HARDWARE CONTROL

As the sbs–4422 board has several different functions, the IOCTL is split into so called "subdevices", one per function. This gives a bit more safety, as the different functions use their own region of registers. So using "subdevices" each function register can be checked for validity. All parts of the 4422–board will be programmed and set up via IOCTL, even all control memory. It would be possible to do the same using "mmap" for registers and on board memory, but unfortunately here is a problem with endianess on board. Together with the hardware control, IOCTL handles several RT–preparing functions, as creating the ring–buffer and starting the I/O.

I/O READ

The sbs_4422 has several options for transferring the actual data to users application. It operates internally with a double buffer method, so presenting a new frame of data in a new buffer. This actual buffer can be mapped into user space, but it will be swapped to new data without notice. In addition, as the sbs_4422 is based on a PLX_9080, the board has its own DMA engine. So it is possible to get an interrupt, as soon as a frame is filled into buffer, start DMA and get another interrupt at DMA–done. Creating a circular buffer in user space and presenting it via IOCTL to the driver, will prepare it for use via DMA. The driver will control the pointers to that buffer and may chain the DMA to fire continuously data into

the ring without user intervention. The information, where the frame of data was stored, is available either via IOCTL or as return value from a read. Now the read will block until new data are received, so freeing from high priority, or return immediately if new data are already avail–able. It is also possible to use "SIGIO" and to react only on that signal, while the DMA runs cyclic through the buffer, but here the way through the system takes more time, than the "read" method.

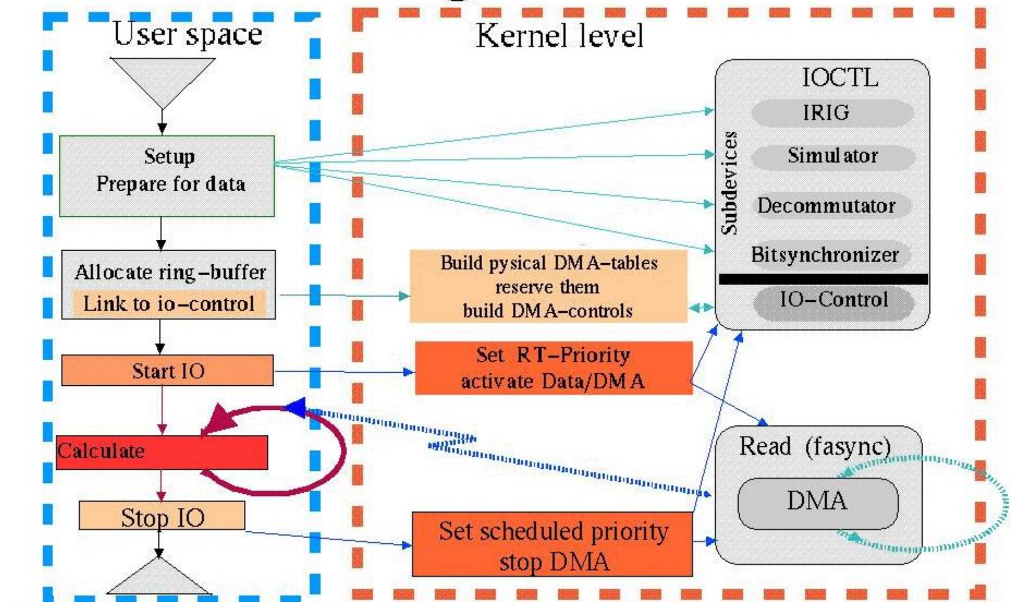
RT–CONTROL

When a program is running and receiving data from sbs_4422, it will loop around a "read" system call. As long, as the data rate won't be to high or the data proc–essing takes to much time, the read will free the cpu at least for some microsec–onds. On the other hand, the task must be ready to process new data when a new frame is received. This behavior is typi–cally for a task scheduled as SCHED_FIFO. But this setting needs superuser privileges. Now the driver switches the process into SCHED_FIFO together with a fixed priority as soon as the start–command (via IOCTL) is given. The priority is calculated in a way, that running more than one board won't use the same level more than once. The stop command (again via IOCTL) will imme–diately force the task back to SCHED_OTHER with default priority.
In addition, at creation of the ring buffer, this memory in user space is reserved as kernel memory to protect it against being swapped out.

Flow of Controls and Data

The picture tries to demonstrate the data and control flow of an application using the sbs–RT–driver. Colors show the increasing priority.

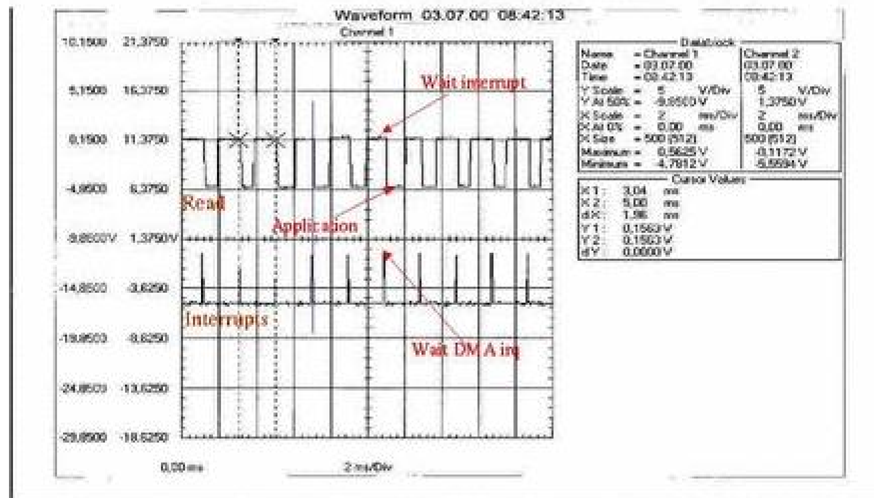
The application starts, setting up the different subdevices, loading the board's memory. As next step, the application allocates enough memory for the ring buffer and presents this buffer via IOCTL to the driver. The driver now generates the DMA tables, saves the physical memory addresses and reserves the buffer as kernel memory, so protecting it against being swapped. From this point on, every preparation is done. So, whenever the

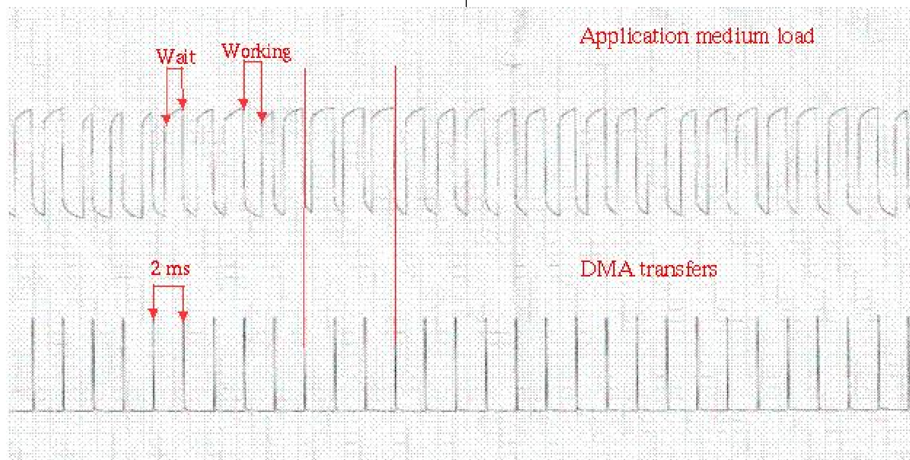


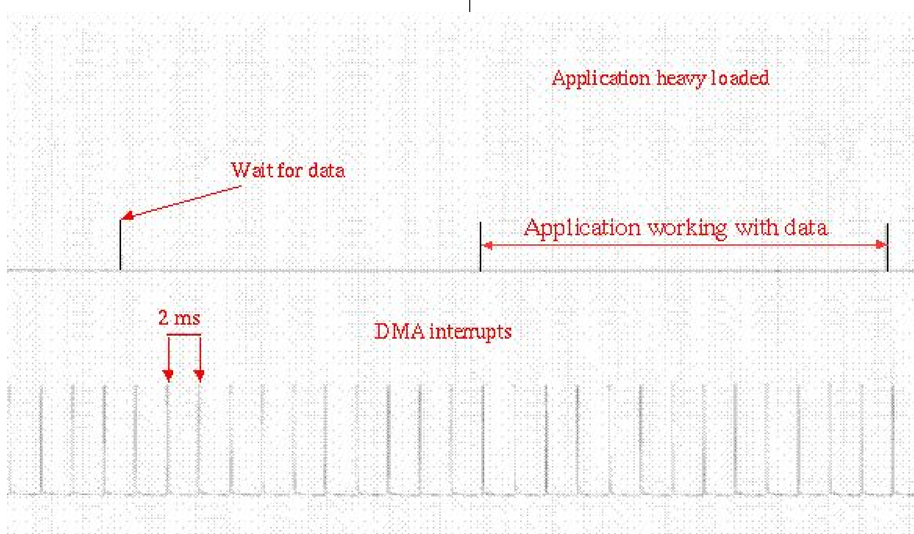
data are ready, the driver is commanded to start data transfer. At this point the appli–cation is forced into RT–priority (SCHED_FIFO) with a high priority. The task itself goes into a loop, using a "read" to get the latest buffer pointer and waits until the next frame of data is stored. During the loop, all sorts of calculations may be done, as well as disc I/O etc.. The task is running in user space and so able to use floating point arithmetic and to use system calls. The application cannot influ–ence the kernel and need not run as root, so protecting the whole system.

Finally, whenever the job should be finished, the data are stopped via an IOCTL call, resetting the priority to normal (SCHED_OTHER, RT–priority 0). Of course this and the release of the ring–buffer is done at "close" as well.

MEASUREMENTS







A sbs–4417 Dac board, needed for the normal telemetry job, was used to get timing measurements. The driver raised a line when entered through the "read"

and lowered it again, when leaving the read. Another line was raised at receiving the "frame" interrupt and lowered when receiving the DMA–done. So plugging a scope at the DAC–outputs an exact meas–uring of timings was possible.

Looking at the printout from the scope, the timing is shown.
The same dac channels were used to get a longer printout of the pulses via paper recorder, showing the stability.

The pulses from interrupts are totally stable at their expected 2 ms rate. The application uses different times for different frames, this is natural, as data are different.

The second diagrams shows clearly, that the time, used for calculation by the appli–cation, is not influencing the behavior of data receiving, only the application itself. The third diagram shows the case, where the application is nearly overloaded. It enters the read sometimes, but most of the time, it tries to work with the old buffers, it runs behind the data, not synchronous anymore. Of course, after a while, some data blocks will be lost. On the other hand, as long as the application is able to keep track, the data are processed synchro–nously in respect to the real time rate on which they are received from the aircraft.

REFERENCES

[1] Alessandro Rubini / Linux Device Drivers
O'Reilly

[2] Linux kernel programming
Addison Wesley

[3] Mbuff
Tomasz Motylowski
moty@stan.chemie.unibus.ch

[4] SBS Technologies, Inc
http://www.sbs.com

[5] PLX Technology
http://www.plxtech.com