 DAVE S.r.l. www.dave.eu		
Written by Matteo Vit, R&D Engineer <i>Dave S.r.l.</i>		VERSION:	1.0.0
		DOCUMENT CODE:	AN-ENG-001
Approved by Andrea Marson, CTO <i>Dave S.r.l.</i>		NO. OF PAGES:	8

AN-ENG-001

Using the AVR32 SoC for real-time video applications

Printed in Italy

Trademarks

Ethernet® is a registered trademark of XEROX Corporation

All other trademarks are the property of their respective owners

Copyright

All rights reserved. Specifications may change any time without notification.

Company Address

DAVE S.r.L.

Via Forniz 2

33080 Porcia (PN) – Italy

Phone: +39 0434 921215

e-mail: info@dave.eu

URL: www.dave.eu

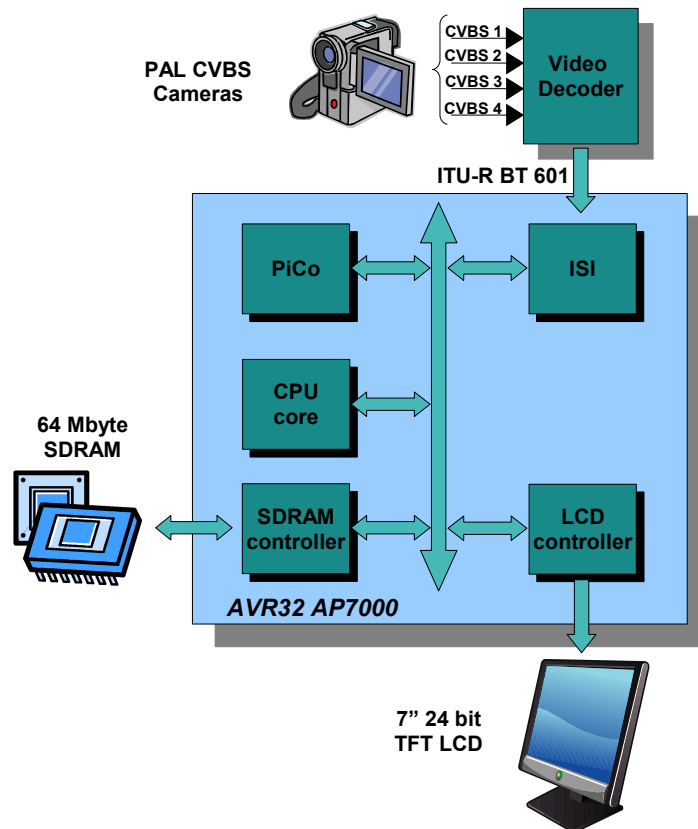
<i>History</i>		
1.0.0	November 2007	First release

Introduction

The AVR32 is a System on Chip (SoC) from Atmel [1] specifically designed for multimedia applications like mobile media player, GPS, hand held devices and so on. The aim of this application note is to show the performance of this platform with real-time video applications. The hardware and the software described in this application note is developed under contract for one of our customers.

Block Diagram

The block diagram below represents the system used for this application note. The operating system is a Linux kernel 2.6.20. A user application opens the video device and copies the frames for displaying them to the LCD panel.



Capturing video from cameras

The camera output is an analog CVBS PAL signal with 625 lines per frame (576 visible lines, the rest being used for other information such as sync data and captioning) and a refresh rate of 50 interlaced fields per second (i.e. 25 full frames per second). This analog signal is then sampled from the video decoder. Its output is an ITU-R BT.601 digital stream [2]. Each frame is composed of 720 x 576 pixels. According to the ITU standard, the two chrominance components

are sampled at half the sample rate of luminance. Since each sample is 8 bits wide, each pixel pair is stored as four bytes: two for luminance, one for blue chroma and one for red chroma. The output digital stream from the video decoder is interlaced like the analog PAL input. This means that the video decoder outputs odd and even lines separately.

AVR32 Image Sensor Interface (ISI)

The AVR32 Image Sensor Interface connects the video decoder to the processor core and to the system SDRAM. The grabbed frames are copied to system memory without CPU intervention through DMA operations. It accepts both YCrCb and RGB inputs up to a resolution of 2048 x 2048. Even if, from the datasheet, the ISI is supposed to work with ITU-R BT.601 streams, it's designed for only progressive devices (like CMOS sensors used on mobile phones and so on). This affected how the device driver is written.

The ISI support both VSYNC/HSYNC synchronization and ITU-R BT.656 embedded SAV/EAV synchronization. In this application note the first synchronization method is used.

Two different video paths are supported: the codec path and the preview path. The former is used for grabbing video frames (i.e. for saving them to disk), the latter for displaying directly to a LCD panel. Inside the preview path there is a scaler that can be used to match the video input size with the LCD one. The same ratio is applied for both horizontal and vertical resize. A color conversion module is also present. The maximum output resolution of the preview path is 640 x 480.

The ISI block can generate an interrupt due to various reasons. The most important is Start Of Frame (SOF). This interrupt is triggered every time a new frame is detected. Because of the VSYNC/HSYNC synchronization method, the SOF interrupt rate is equal to the VSYNC frequency.

Video for Linux (V4L2) driver

Video for Linux is the capture/overlay API for Linux kernels. The advantage of using a standard API is that every video device driver and application compliant with it can be used together. For example you can use the driver with a custom application or mainstream media player/encoder like mplayer/mencode.

The driver for the ISI used in this application note is based on the Corbet's `cafe_ccic` driver (for the One Laptop Per Child - OLPC project).

Before describing how the device driver works, a few trade-offs need to be explained. As previously stated, the video decoder output is a frame of 720 x 576 pixels with 50 interlaced fields per second. This means that the first 720 x 288 (576 / 2) pixels field contains odd lines only, the second one contains even lines only and so on. Since the ISI doesn't provide any support for interlaced input, there is no way to know if a frame contains even or odd lines. With this constraints the driver actually grabs only one field and discards the next one. A software decimation factor of 2 is applied on each line to obtain a standard 360 x 288 pixels format. The

scaler of the preview path (and the preview path itself) isn't used since the maximum output is 640 x 480 pixels and only symmetrical resize ratios can be used.

From the V4L perspective, the driver can support both MMAP and read I/O method. After the device is open, the application can obtain information about the format supported (360 x 288 YUV). The ISI registers are written with video input resolution and the codec path is started. A ring of frame buffer is used for storing the image captured from the ISI. If the application performs MMAP I/O method another ring of buffer is used for frames interchange between kernel and user space.

Displaying video

After the video stream is grabbed from the V4L2 driver, it's displayed on the 7" 480 x 234 LCD panel. Before doing this, a color space conversion is required because the video decoder output is YUV format while the LCD format is RGB. The matrix below is used for the color space conversion.

$$\begin{bmatrix} R \\ G \\ B \end{bmatrix} = \begin{bmatrix} C_{00} & C_{01} & C_{02} \\ C_{10} & C_{11} & C_{12} \\ C_{20} & C_{21} & C_{22} \end{bmatrix} \begin{bmatrix} Y \\ U \\ V \end{bmatrix} + \begin{bmatrix} O_0 \\ O_1 \\ O_2 \end{bmatrix}$$

Since input and output pixels are 8 bit integer values, an optimized version can be used:

```
R = clip(( 298 * C          + 409 * E + 128) >> 8)
G = clip(( 298 * C - 100 * D - 208 * E + 128) >> 8)
B = clip(( 298 * C + 516 * D          + 128) >> 8)
```

The color space converter inside the preview path can't be used since its maximum output is 640 x 480 pixels, while it's supposed to be 720 x 288 for the analysis done below.

Color conversion with AVR32 Pixel Coprocessor (PiCo)

The color conversion is a computational complex task. Even if 8 bit integer math is used, for each RGB pixel seven MACs (multiplication and accumulation) are required (two matrix coefficients are zero). This means 725760 MACs for a 360 x 288 pixels frame. Using this implementation, a performance of 14 fps with 100% CPU load is achieved (with AT32AP7000 @ 140 MHz). Of course, a system with a 100% of CPU load can't be used for doing anything else useful. The next step is to use the internal AVR32 pixel coprocessor, PiCo, to speed up the color conversion process.

The Pixel Coprocessor (PiCo) consists of three Vector Multiplication Units (VMU), an Input Pixel Selector and Output Pixel Inserter. Each VMU can perform a vector multiplication of a 1x3 12-bit coefficient vector with 3x1 8-bit pixel vector plus a 12-bit offset. The Input Pixel Selector is used to choose which pixel enters the VMU. The Output Pixel Inserter is used to store pixels in packed or planar format. The PiCo is composed of three pipeline stages with a throughput of one operation per CPU clock cycle. It can be used to accelerate color conversion due to the matrix like operations involved.

The color conversion routine loads the PiCo matrix coefficients first. Four pixel (eight bytes) at time are loaded inside the PiCo registers. Due to the chroma subsampling, two RGB pixels are computed using the same U and V value, but with adjacent Y values. Before storing the RGB pixels, four new pixels are loaded inside PiCo registers. This is used to keep full the PiCo pipeline and thus to achieve more throughput. The RGB pixels are stored in packet format directly inside the frame buffer.

```

Enabling debugfs
* Opening frame buffer ...
* Mmapping frame buffer ...
mmapFB x = 480, y = 234, bpp = 24
* Init PiCo ...
* Opening isi ...
* Initializing isi ...
* Start capturing ...
* Main loop ...
frame rate 25.022520 fps
frame rate 25.002178 fps
frame rate 25.002711 fps
frame rate 25.001405 fps
frame rate 25.002203 fps
frame rate 25.002649 fps
frame rate 25.001547 fps
frame rate 25.001999 fps
frame rate 25.002630 fps
frame rate 25.002463 fps
frame rate 25.002079 fps
frame rate 25.002432 fps
frame rate 25.001751 fps
frame rate 25.002005 fps
* Stop capturing ...
* Uninitialize isi ...
* Closing isi ...
* Unmapping FB ...
Capture done

```

Box 1: The demo application running at 25 fps

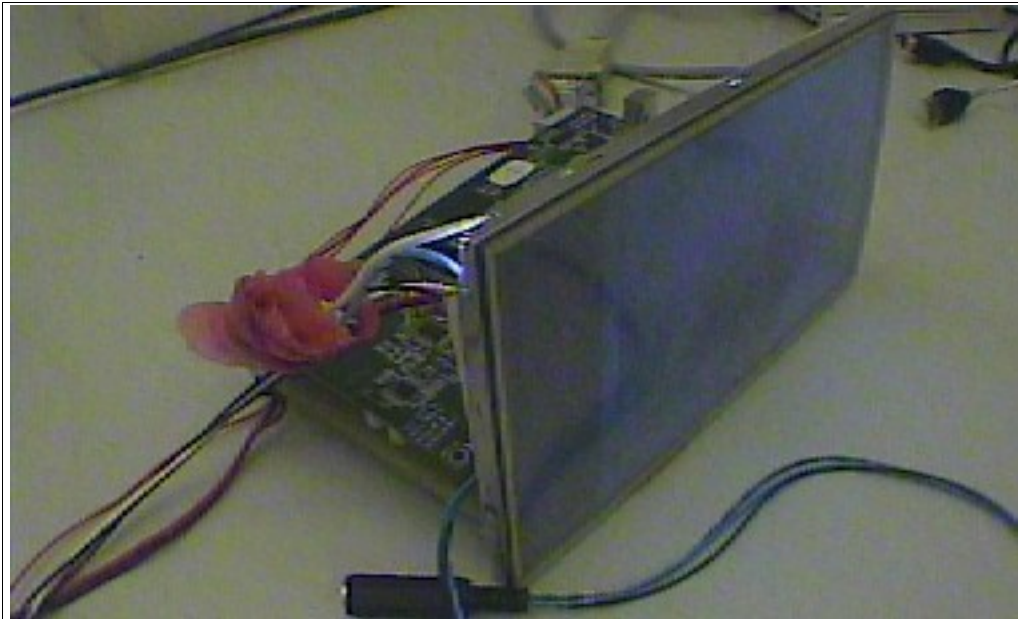
Using the PiCo for color space conversion, a performance of 25 fps (full frame rate) with only 27% CPU load is achieved (with the same AT32AP7000 @ 140 MHz).

```

192.168.0.103 - PuTTY
Mem: 9160K used, 53100K free, 9160K shrd, 0K buff, 2092K cached
Load average: 0.15 0.04 0.01 (Status: S=sleeping R=running, W=waiting)
  PID USER      STATUS  RSS   PPID  %CPU  %MEM  COMMAND
  293 root      S       624   289  27.3   0.9  test_hisi
  288 root      R       340   287   1.8   0.5  top
  250 root      S       144     1   0.3   0.2  telnetd
  286 root      S       424     1   0.0   0.6  sh
  287 root      S       392   250   0.0   0.6  ash
  289 root      S       332   286   0.0   0.5  display.sh
    1 root      S       312     0   0.0   0.4  init
  242 root      S       228     1   0.0   0.3  dnsmasq
  272 root      S       216     1   0.0   0.3  dropbear
  257 root      S       204     1   0.0   0.3  inetd
  181 root      S       176     1   0.0   0.2  syslogd
  191 root      S       164     1   0.0   0.2  klogd
  264 root      S       112     1   0.0   0.1  httpd
  146 root      SW<     0      6   0.0   0.0  rpciod/0
    5 root      SW<     0      1   0.0   0.0  khelper
    2 root      SWN     0      1   0.0   0.0  ksoftirqd/0
    3 root      SW      0      1   0.0   0.0  watchdog/0
    4 root      SW<     0      1   0.0   0.0  events/0
    6 root      SW<     0      1   0.0   0.0  kthread
   53 root      SW<     0      6   0.0   0.0  kblockd/0
   56 root      SW<     0      6   0.0   0.0  khubd

```

Box 2: CPU load



Box 3: Frame buffer screenshot

Conclusion

The results presented in this application note are impressive. A 360 x 288 full frame rate video stream is grabbed from the video decoder and then displayed on LCD using only the 27% of the AVR32 CPU load at 140 MHz. This means that there is room for other tasks (like user interface, other I/O operations, and so on) using a low clock frequency and thus a low power consumption. Before the AVR32 SoC, real-time video applications can be done only using high end ARM processor (like ARM11) or dedicated DSP coprocessor.

References

- 1: Atmel, <http://www.atmel.com>
- 2: ITU, <http://www.itu.int>