Merry Christmas and Happy New Year 2002!
Brought to you by ST Micro and Xilicon!
Greetings to StuntGuy and his team!
Have fun and don't forget to post your findings :)

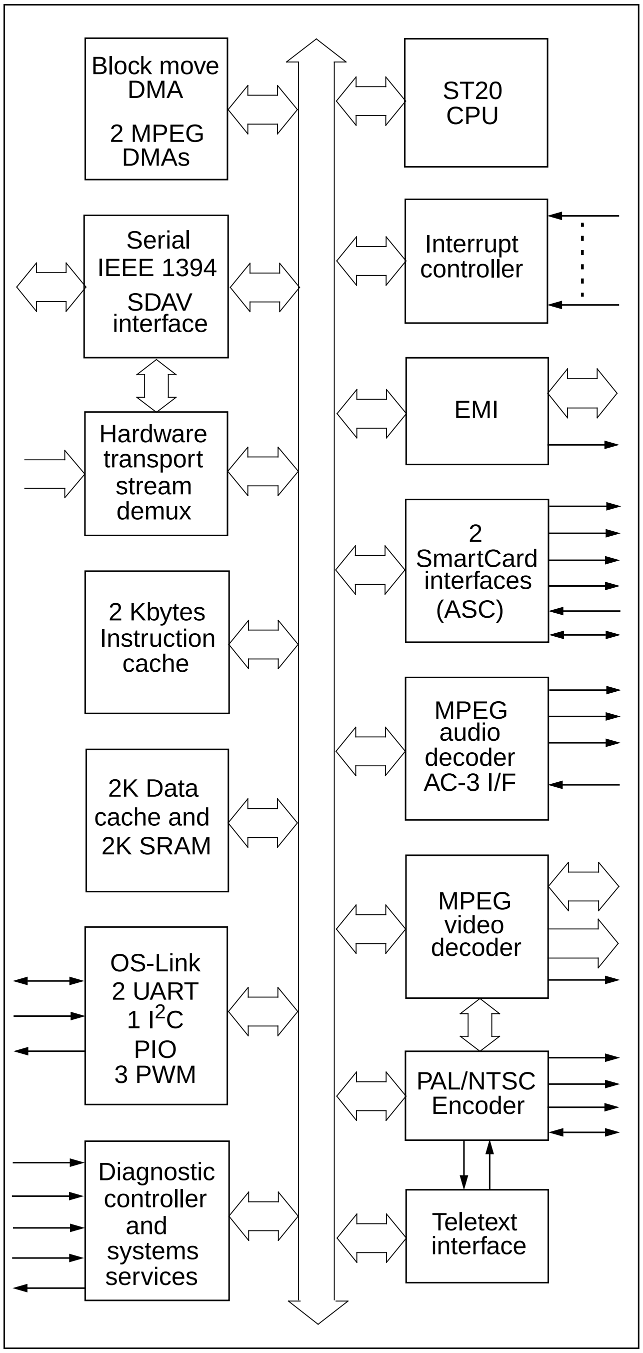
# STi5500

## SET TOP BOX BACKEND DECODER WITH INTEGRATED HOST PROCESSOR

**PRELIMINARY DATA**

### FEATURES

- Enhanced 32-bit VL-RISC CPU - 50 MHz clock
  - fast integer/bit operation and very high code density
- High performance memory/cache subsystem
  - 2 Kbytes Instruction cache, 2K bytes SRAM, 2 Kbytes data cache/SRAM
  - 200 Mbytes/s maximum bandwidth
- Combined video and audio decoder core
  - Video decoder fully supports MPEG-2 MP@ML. Letter box filter
  - Memory reduction - PAL MP@ML in 12 MBits
  - 2 to 8 bit per pixel OSD options
  - Audio decoder supports layers 1 and 2 of MPEG, interface to external AC-3 decoder.
- PAL/NTSC encoder
  - Macrovision, teletext, and closed caption.
  - Outputs RGB and CVBS, Y, C
- High performance SDRAM memory interface
  - Supports 1 or 2 16-Mbit 100 MHz SDRAMs
  - Accessible by MPEG decoder, CPU and DMAs
  - High bandwidth access from CPU allows high performance OSD operations
- Programmable memory interface
  - 4 banks each 8/16 bits wide
  - Support for mixed memory, peripherals and DRAM
- Hardware transport stream demultiplexor
  - Serial input
  - Supports DSS, DVB, and DVD bit streams
  - 32 PIDs supported
  - DES and DVB descramblers
- Vectored interrupts - 8 prioritized levels.
- DMA engines/interfaces
  - 2 SmartCard interfaces, 2 UARTs, 1 I$^2$C controller, 3 PWM outputs, 3 timers, 3 capture timers.
  - 34 bits of PIO shared with serial interfaces
  - OS-Link interface
  - Block move DMA, 2 MPEG DMAs
  - Teletext interface
  - Serial or 1394 A/V link layer interface
- Professional toolset support
  - ANSI C compiler and libraries
  - INQUEST advanced debugging tools
- Non-intrusive debug controller
  - Hardware breakpoints
  - Real time trace
- JTAG Test Access Port
- 208 pin PQFP package


Block move DMA
2 MPEG DMAs
ST20 CPU
Serial IEEE 1394
SDAV interface
Interrupt controller
EMI
Hardware transport stream demux
2 SmartCard interfaces (ASC)
2 Kbytes Instruction cache
MPEG audio decoder AC-3 I/F
2K Data cache and 2K SRAM
MPEG video decoder
OS-Link
2 UART
1 I$^2$C
PIO
3 PWM
PAL/NTSC Encoder
Diagnostic controller and systems services
Teletext interface

### APPLICATIONS

- Set Top Boxes to DVB and DSS standards

6/4/99

7110597 A

The information in this data sheet is subject to change

# TABLE OF CONTENTS

## Part E Peripherals 228

## Part F Timing and electrical data 266

# 1 Introduction

The STi5500 is a programmable transport and MPEG decoder IC designed to meet the specifications for DVB and DSS set top box systems.

The STi5500 combines the functionality of the set top box transport IC, system microcontroller, audio and video MPEG decoders, and the PAL/NTSC encoder into a single device.

Transport functions are performed in a hardware module. This block can be programmed to process bitstreams for the two standards and includes DES and DVB descramblers and SI filtering.

The performance offered by the ST20 32-bit micro-core allows the following operations to be performed in software:

1 Device drivers and synchronization,

2 System management functions,

3 Electronic program guide,

4 Conditional access module.

The use of a 32-bit CPU enables advanced graphics routines to be employed for on-screen display functions, allowing fast turnaround system upgrades.

The ST20 micro-core family has been developed by STMicroelectronics to provide the tools and building blocks to enable the development of highly integrated application-specific 32-bit devices at the lowest cost and fastest time to market. The ST20 macrocell library includes the ST20Cx family of 32-bit VL-RISC (variable length reduced instruction set computer) micro-cores, embedded memories, standard peripherals, I/O, controllers and ASICs.

The STi5500 uses the ST20 macrocell library to provide all of the dedicated hardware modules required in a DVB/DSS set top box programmable transport-IC. These include:

- High performance internal SRAM and cache subsystem,
- I$^2$C interface to other devices in the set top box,
- UART serial I/O interface to modem and auxiliary ports,
- Interrupt controller for internal and external interrupts,
- DMA to MPEG audio and video device(s),
- External memory interface supporting DRAM, EPROM and peripherals,
- PWM/timer module for control of system clock VCXOs,
- Programmable I/O pins,
- Smart card interfaces.

The STi5500 also integrates the functions of the MPEG decoder and PAL/NTSC encoder with the following features:

- Video decoder fully supports MPEG-2 Main Profile/Main Level (MP@ML).
- Memory reduction architecture allows sharing of single 16-Mbit SDRAM between MPEG decoding, micro and transport functions - memory expandable to 32 Mbits of SDRAM.
- Letter box filter.
- 2-bit to 8-bit OSD.
- PAL/NTSC encoder.
- RGB outputs.
- CVBS, Y, C outputs.
- Close caption.
- Macrovision 7.01.
- Teletext insertion.
- 2-channel MPEG audio decoder with interface to external audio decoders.

The STi5500 has been designed to minimize system costs. The external memory interface contains a zero glue logic DRAM controller and a low-cost 16-bit EPROM interface. The SDRAM memory interface directly supports 100MHz SDRAMs providing the very high bandwidths to support MPEG decoding and display, OSD drawing and display, and general system use. Furthermore the ST20 VL-RISC micro-core has the highest code density of any 32-bit CPU, leading to the lowest cost program ROM.

This data sheet refers to silicon versions D and onwards. The version letter can be identified from the label on the package; the label is of the form:

*X-VYY*

where *V* is the main version letter.

Changes to this data sheet since the last edition (42-1696-02 dated July 1998) are marked with change bars and listed in

# Part A Architecture

# 2 STi5500 architecture overview

A block diagram of a digital set top receiver based on the ST5500 is shown in Figure 2.1.
A DVB receiver is illustrated, however a DSS receiver would be very similar.
The STi5500 performs the system microcontroller, transport demultiplexer, MPEG video and MPEG audio decoders, as well as the PAL/NTSC encoder functions. It has been designed to directly interface with external memory and peripherals with no extra glue logic, keeping the system cost to a minimum. The STi5500 architectural block diagram is shown on the front cover.

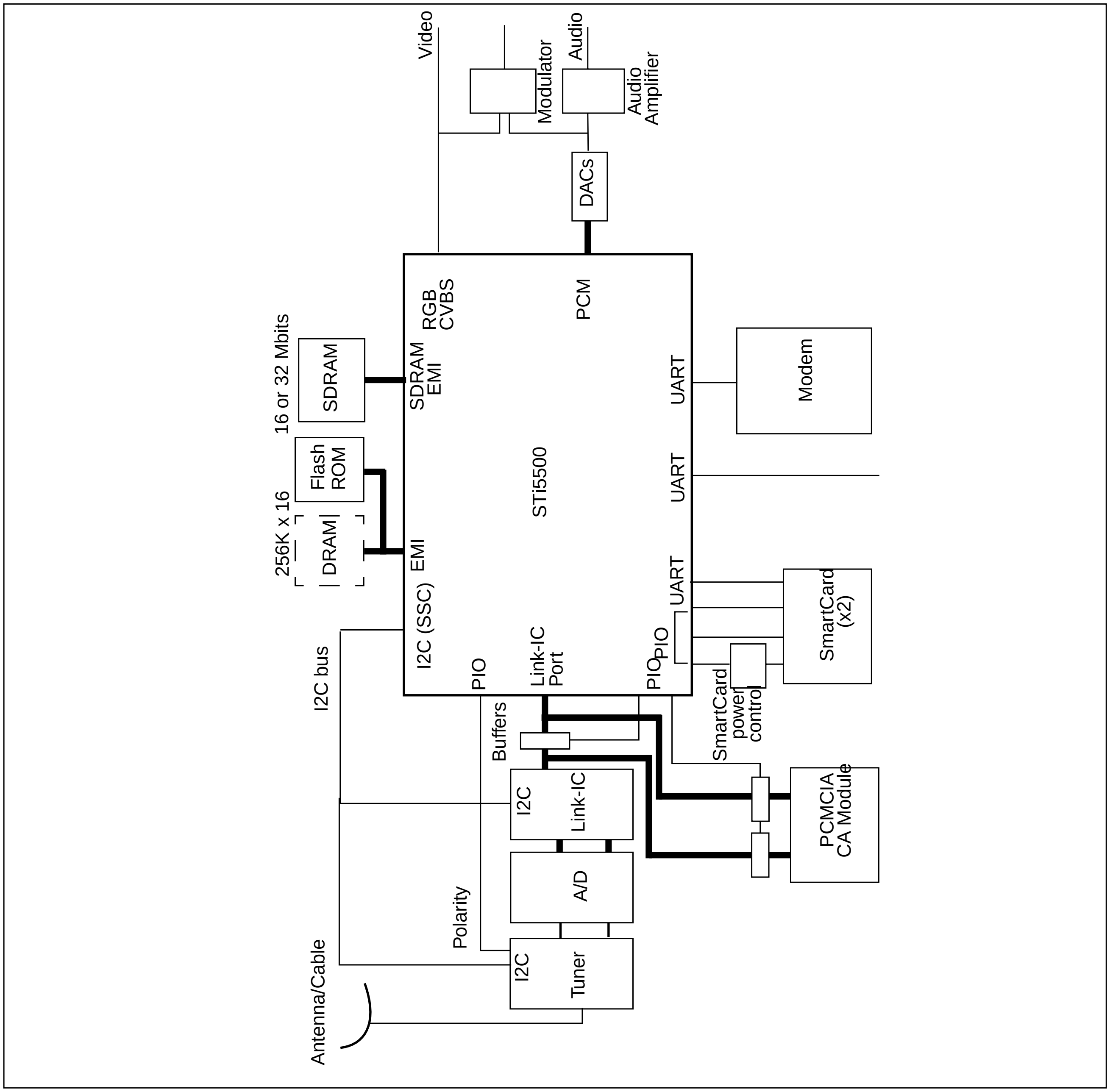


Figure 2.1 DVB Digital set top box block diagram

## 2.1 STi5500 functional modules

The front cover shows the subsystem modules that comprise the STi5500. These modules are outlined below and more detailed information is given in the following chapters of this datasheet.

### 2.1.1 ST20 and peripherals

**CPU**

The Central Processing Unit (CPU) on the STi5500 is the ST20-C2 32-bit processor core. It contains instruction processing logic, instruction and data pointers and an operand register. It directly accesses the high speed on-chip SRAM memory, which can store data or programs, and uses caches to reduce access time to off-chip program and data memory. The processor can access memory via the general purpose External Memory Interface (EMI) or via the SDRAM EMI which is shared with the MPEG decoder.

**Memory subsystem**

The STi5500 on-chip SRAM memory system provides 200 Mbytes/s internal data bandwidth, supporting pipelined 2-cycle internal memory access at 20 ns cycle times. The STi5500 memory system consists of 2 Kbytes of SRAM, 2Kbytes of instruction cache, a 2Kbyte data cache that can be programmed to be SRAM, and an external memory interface (EMI).

The STi5500 product has 2 Kbytes of on-chip SRAM. The advantage of this is the ability to store time critical code on chip, for instance interrupt routines, software kernels or device drivers, and even frequently used data without these being flushed from the caches.

The instruction and data caches are direct mapped with a write-back system for the data cache and support burst accesses to the external memories for refill and write-back which are effective for increasing performance with page-mode and SDRAM memories.

The STi5500 EMI controls access to the external memory and peripherals while the SDRAM EMI provides access to the SDRAM buffer for the MPEG decoders, ST20 and DMA peripherals.

The STi5500 EMI can access a 16 Mbyte (or greater if DRAM is used) physical address space in each of the four general purpose memory banks, and provides sustained transfer rates of up to 80 Mbytes/s. Peripherals that support an asynchronous data acknowledge are supported.

High memory bandwidths up to 200 Mbytes/s can be supported by the SDRAM EMI.

The STi5500 internal memory interconnect provides buffering and arbitration of memory access requests to sustain very high throughput of memory accesses.

### System services module

The STi5500 system services module includes:

- phase locked loop (PLL) - accepts 27 MHz input and generates all the internal high frequency clocks needed for the CPU and the OS-Link.
- test access port - JTAG compatible.
- Diagnostics controller accessed via the JTAG port providing:
  - Bootstrapping during development
  - Hardware breakpoint and watchpoint
  - Real time trace
  - External LSA triggering support.

### Serial communications

To facilitate the connection of this system to a modem for a pay-per-view type system and other peripherals, two UARTs (ASCs) are included in the device. The UARTs provide an asynchronous serial interface. The UART can be programmed to support a range of baud rates and data formats, for example, data size, stop bits and parity.

One synchronous serial communications (SSC) interface is provided on the device. This can be used to control the Link-IC and the remote control devices in the application via an $I^2C$.

### Interrupt subsystem

The STi5500 interrupt subsystem supports eight prioritized interrupt levels. Three external interrupt pins are provided. Level assignment logic allows any of the internal or external interrupts to be assigned, and if necessary share, any interrupt level.

### Transport stream demultiplexor

The transport stream demultiplexing function is performed by a hardware on-chip module. The transport stream demultiplexor is sometimes called the link interface, since it interfaces to the Link IC.

Data packets from the Link-IC input interface are input into a FIFO while the PID is checked to see if it is currently selected for processing or is to be discarded. A selected packet is parsed by the module to determine its type and to extract data from it. If the packet is encrypted the correct key is written into the correct decryption core in the transport stream demultiplex module and the packet is decrypted.

After parsing and descrambling the packet, the data is either transferred to buffers in external memory or directly to the MPEG audio and video decoders. If the audio and video data is buffered then the data can be DMA transferred from the buffer to the MPEG decoders.

Transport packets with up to 32 different PIDs can be extracted. A second filter function is applied to all section data. The maximum length of the filters is 16 bytes for DSS and 14 bytes for DVB.

Error conditions, system time clock recovery, and control of the hardware module are handled by software running on the CPU.

### SmartCard interfaces

The SmartCard interfaces support SmartCards that are compliant with ISO7816-3 and use the asynchronous protocol.

### PWM and counter module

This unit includes three separate pulse width modulator (PWM) generators using a shared counter, and three timer compare and capture channels sharing a second counter.

The counters can be clocked from a pre-scaled internal clock or from a pre-scaled external clock via the capture clock input and the event on which the timer value is captured is also programmable.

The PWM counters are 8-bit with 8-bit registers to set the output high time. The capture/compare counter and the compare and capture registers are 32-bit.

### Parallel IO module

34 bits of parallel IO are provided. Each bit is programmable as an output or an input. The output can be configured as a totem pole or open drain driver. Input compare logic is provided which can generate an interrupt on any change on any input bit.

Many pins of the STi5500 device are multi-function and can either be configured as PIO or connected to an internal peripheral signal.

### Teletext

Teletext data is read from a user defined external memory buffer using a dedicated DMA, and is encoded in the “World System Teletext” format in the composite video signal.

## 2.1.2 MPEG decoder subsystem

This subsystem takes the MPEG compressed data streams and decompresses them outputting digital YUV data in the case of the video and stereo PCM samples in the case of the audio decoder.

An interface is provided to output an audio bitstream for decoding by an external MPEG or AC-3 decoder to support multi-channel (surround) audio.

The digital video data is fed to the PAL/NTSC encoder subsystem.

The subsystem includes support for on-screen display (OSD) graphics which can be programmed to be mixed with the digital video output from the video decoder.

## 2.1.3 PAL/NTSC encoder

Integrated into this subsystem is all of the digital processing and the digital to analog convertors required to process the digital video output from the MPEG video decoder and produce RGB and CVBS analog outputs. The output of the teletext interface is filtered and re-inserted into the blanking interval in this subsystem. The Macrovision Anti-taping system is supported.

# 3 Pin list

## 3.1 Pin Functions

Signal names are prefixed by **not** if they are active low, otherwise they are active high.

### Supplies

| Pin | Number | Function |
|---|---|---|
| **Vdd** | 19 | Power supply. |
| **Gnd** | 18 | Ground. |
| **Vdda** | 2 | Analog power supply for PAL/NTSC encoder. |
| **Vssa** | 2 | Analog ground for PAL/NTSC encoder. |

Table 3.1 Supply pins

### Transport stream demultiplexor

| Pin | In/Out | Function |
|---|---|---|
| **F_B_Clk** | in | FEC bit clock. |
| **F_Data** | in | FEC serial data. |
| **F_Error / F_P_Start** | in | Link error (DVB/DSS) or channel packet start (DVD). |
| **F_P_Clk / F_D_Valid** | in | Link packet clock (DVB/DSS) or channel data valid (DVD). |
| **Link_Ext_Clk** | in | External clock to the transport stream demultiplexor block. |

Table 3.2 Transport stream demultiplexor pins

### MPEG1 audio output and AC-3 audio interface

| Pin | In/Out | Function |
|---|---|---|
| **Pcm_Data / A_C_Data** | out | PCM data out or AC-3 data. |
| **Pcm_ClkIn** | in/out | PCM clock input from VCXO. |
| **Pcm_ClkOut / A_C_Stb** | out | PCM clock out or AC-3 data strobe. |
| **A_C_Req** | in | AC-3 audio data request. |
| **A_Pts_Stb** | in | AC-3 audio PTS strobe. |
| **LrClk / A_Word_Clk** | out | Left/right channel clock or AC-3 word clock. |

Table 3.3 MPEG1 audio output and AC-3 audio interface pins

### Video output interface

| Pin | In/Out | Function |
|---|---|---|
| **R_Out** | out | Red output. |
| **G_Out** | out | Green output. |
| **B_Out** | out | Blue output. |
| **C_Out** | out | Chroma output. |
| **CV_Out** | out | Composite video output. |
| **Y_Out** | out | Luma output. |
| **I_Ref_Dac_RGB** | in | DAC current reference. |
| **I_Ref_Dac_YCC** | in | DAC current reference. |
| **V_Ref_Dac_RGB** | in | DAC voltage reference. |
| **V_Ref_Dac_YCC** | in | DAC voltage reference. |
| **Osd_Active** | in/out | OSD active. |
| **PixClk_27Mhz** | in | STi5500 system clock from VCXO. |
| **not_Hsync** | in/out | Horizontal sync. |
| **Odd_not_Even** | in/out | Vertical sync. |

Table 3.4 Video output interface pins

### SDRAM interface

| Pin | In/Out | Function |
|---|---|---|
| **Ad0-11** | out | SDRAM address bus. |
| **Dq0-15** | in/out | SDRAM data. |
| **not_SdCS0-1** | out | SDRAM chip selects. |
| **not_SdCas** | out | SDRAM CAS. |
| **not_SdRas** | out | SDRAM RAS. |
| **not_SdWE** | out | SDRAM write enable. |
| **MemClkIn** | in | SDRAM memory clock input. |
| **MemClkOut** | out | SDRAM memory clock output. |
| **Dqml** | out | DQ mask enable (lower). |
| **Dqmu** | out | DQ mask enable (upper). |

Table 3.5 SDRAM Interface pins

### External memory interface

| Pin | In/Out | Function |
|---|---|---|
| **Adr1-21** | out | External memory address bus. |
| **Data0-15** | in/out | External memory data bus. |
| **not_Ras1** | out | DRAM RAS. |
| **ReadnotWrite** | out | DRAM R/W strobe. |
| **MemWait** | in | Memory cycle wait. |
| **not_WE0-1** | out | Byte enable. |
| **not_Cas0** | out | DRAM CAS. |
| **not_Cas1** | out | DRAM CAS. |
| **not_CE1-3** | out | Chip select for banks 1 - 3. |
| **not_Ras0 / not_CE0** | out | DRAM RAS or chip select for bank 0. |
| **not_OE** | out | Output enable of RAM / ROM. |
| **ProcClockOut** | out | Processor clock. |

Table 3.6 EMI pins

### External interrupts

| Pin | In/Out | Function |
|---|---|---|
| **Irq0-2** | in | Modem, AC-3, servo interrupts. |

Table 3.7 External interrupt pins

### NRSS serial interfaces

| Pin | In/Out | Function |
|---|---|---|
| **Nrss_Clk** | out | NRSS serial clock. |
| **Nrss_In** | in | NRSS serial data input. |
| **Nrss_Out** | out | NRSS serial data output. |

Table 3.8 NRSS serial interface pins

### Programmable I/O

| Pin | In/Out | Function |
|---|---|---|
| **Pio0_0/1, Pio0_3-7** | in/out | General purpose IO. |
| **Pio1_0/1, Pio1_2-7** | in/out | General purpose IO. |
| **Pio2_0/1, Pio2_3-5, Pio2_7** | in/out | General purpose IO. |
| **Pio3_0-7** | in/out | General purpose IO. |
| **Pio4_0-7** | in/out | General purpose IO. |

Table 3.9 Programmable I/O pins

The alternative functions of the PIO pins and the shared pins are described in section 3.2.

### JTAG interface

| Pin | In/Out | Function |
|---|---|---|
| **Tck** | in | Test clock. |
| **Tdi** | in | Test data input. |
| **Tdo** | out | Test data output. |
| **Tms** | in | Test mode select. |
| **not_Trst** | in | Test reset. |

Table 3.10 JTAG interface pins

### System use

| Pin | In/Out | Function |
|---|---|---|
| **Brm2** | out | PWM output 2. |
| **Brm1 / BootFromRom** | out/in | PWM output 1 or boot from ROM during reset. |
| **Brm0 / Oslink_Sel** | out/in | PWM output 0 or configure OS-Link pins. |
| **not_Rst** | in | STi5500 reset. |

Table 3.11 System pins

### SDAV bus interface / P1394 bus interface

| Pin | In/Out | Function |
|---|---|---|
| **Sdav_Clk / P1394_Clk** | in/out | SDAV data strobe/clock or P1394 clock. |
| **Sdav_Data** | in/out | SDAV data line. |
| **Sdav_Dir / P1394_P_Clk** | in/out | SDAV data direction or P1394 packet clock. |
| **Osc_In / 27Mhz_Out** | in/out | SDAV crystal input (49.152MHz) or 27MHz out. |

Table 3.12 SDAV and P1394 bus interface pins

## 3.2 PIO pins and alternative functions

To improve flexibility and to allow the STi5500 to fit into different set-top box application architectures, the input and output signals from some of the peripherals are not directly connected to the pins of the device. Instead they are assigned to the alternative function inputs and outputs of a PIO port bit. This scheme allows these pins to be configured as general purpose PIO if the associated peripheral input or output is not required in that particular application.

Peripheral inputs connected to the alternative function input of a PIO bit are permanently connected to the input pin. The output signal from a peripheral is only connected when the PIO bit is configured into either push-pull or open drain driver alternative function mode.

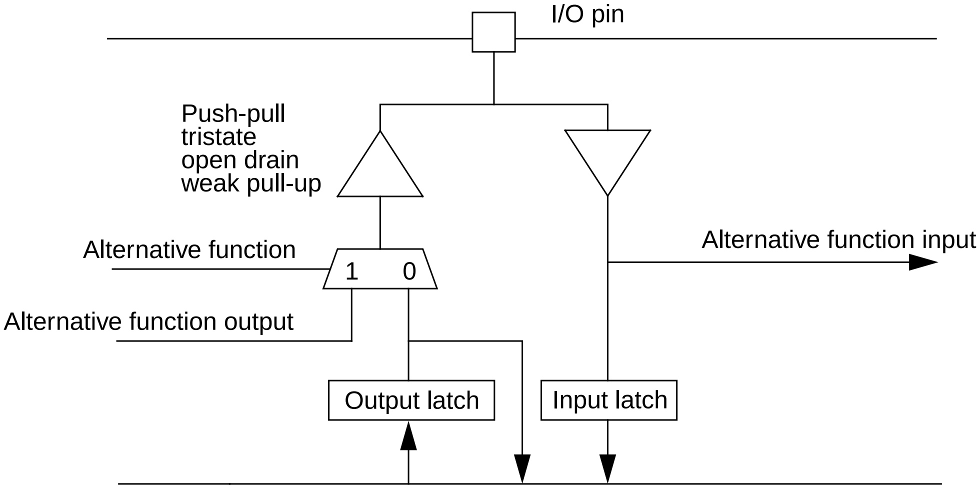


Figure 3.1 I/O port pin

Table 3.13 shows the assignment of the alternative functions to the PIO bits. Parentheses ( ) in the table indicate suggested or possible pin usages as a PIO, not an alternative function connection.

| Port bit | Alternative function of PIO pins | | | | |
|---|---|---|---|---|---|
| | **PIO port 0** | **PIO port 1** | **PIO port 2** | **PIO port 3** | **PIO port 4** |
| 0 | **ASC0TxD or Sc1DataOut** | **SSC0 MTSR** | **ASC2TxD or Sc0DataOut** | | |
| 1 | **ASC0RxD or Sc1DataIn** | **SSC0 MRST** | **ASC2RxD or Sc0DataIn** | | |
| 2 | Not connected | **SSC0 SClk** | Not connected | | |
| 3 | **Sc1Clk** | **CaptureIn1** | **Sc0Clk** | | |
| 4 | **(Sc1RST)** | **CaptureIn2** | **CompareOut0 (Sc0RST)** | | |
| 5 | **(Sc1CmdVcc)** | **ASC1TxD** | **(Sc0CmdVcc)** | | **CompareOut1 (IROut)** |
| 6 | **ASC0Dir** | **ASC1RxD** | **Not connected** | **TriggerIn** | **CaptureIn3** |
| 7 | **(Sc1Detect)** | **ASC3TxD** | **CaptureIn0 (Sc0Detect)** | **TriggerOut** | **ASC3RxD** |

Table 3.13 Alternative function of PIO pins

Pins **Pio0_0/1**, **Pio1_0/1** and **Pio2_0/1** are shared between bit 0 and bit 1 of their respective ports. The different functions for output and input are shown in Table 3.14.

| Pin | Output function (port bit) | Input function (port bit) |
|---|---|---|
| **Pio0_0/1** | **Pio0_0** | **Pio0_1** |
| **Pio1_0/1** | **Pio1_0** | **Pio1_1** |
| **Pio2_0/1** | **Pio2_0** | **Pio2_1** |

Table 3.14 Functions of shared PIO pins

# 4 Package specification

The STi5500 is available in a 208 pin plastic quad flat pack (PQFP) package.

## 4.1 208 pin PQFP package dimensions

| REF. | CONTROL DIM. mm | | | ALTERNATIVE DIM. INCHES | | | NOTES |
|---|---|---|---|---|---|---|---|
| | MIN | NOM | MAX | MIN | NOM | MAX | |
| A | - | - | 4.080 | - | - | 0.161 | |
| A1 | 0.25 | - | 0.40 | 0.010 | - | 0.016 | |
| A2 | 3.240 | 3.600 | 3.740 | 0.127 | 0.142 | 0.147 | |
| B | 0.190 | - | 0.380 | 0.007 | - | 0.015 | |
| C | 0.120 | - | 0.180 | 0.005 | - | 0.007 | |
| D | 30.350 | - | 30.850 | 1.195 | - | 1.215 | |
| D1 | 27.900 | 28.000 | 28.100 | 1.098 | 1.102 | 1.106 | |
| D3 | - | 25.500 | - | - | 1.004 | - | REF |
| E | 30.350 | - | 30.850 | 1.195 | - | 1.215 | |
| E1 | 27.900 | 28.000 | 28.100 | 1.098 | 1.102 | 1.106 | |
| E3 | - | 25.500 | - | - | 1.004 | - | REF |
| e | - | 0.500 | - | - | 0.020 | - | BSC |
| K | 0 | - | 7 | 0 | - | 7 | |
| L | 0.350 | 0.500 | 0.650 | 0.014 | 0.020 | 0.026 | |
| L1 | - | 1.300 | - | - | 0.051 | - | TYP |
| Zd | - | 1.250 | - | - | 0.049 | - | REF |
| Ze | - | 1.250 | - | - | 0.049 | - | REF |

Table 4.1 208 pin PQFP package dimensions

**Notes**

1 Lead finish to be 85 Sn/15 Pb solder plate.

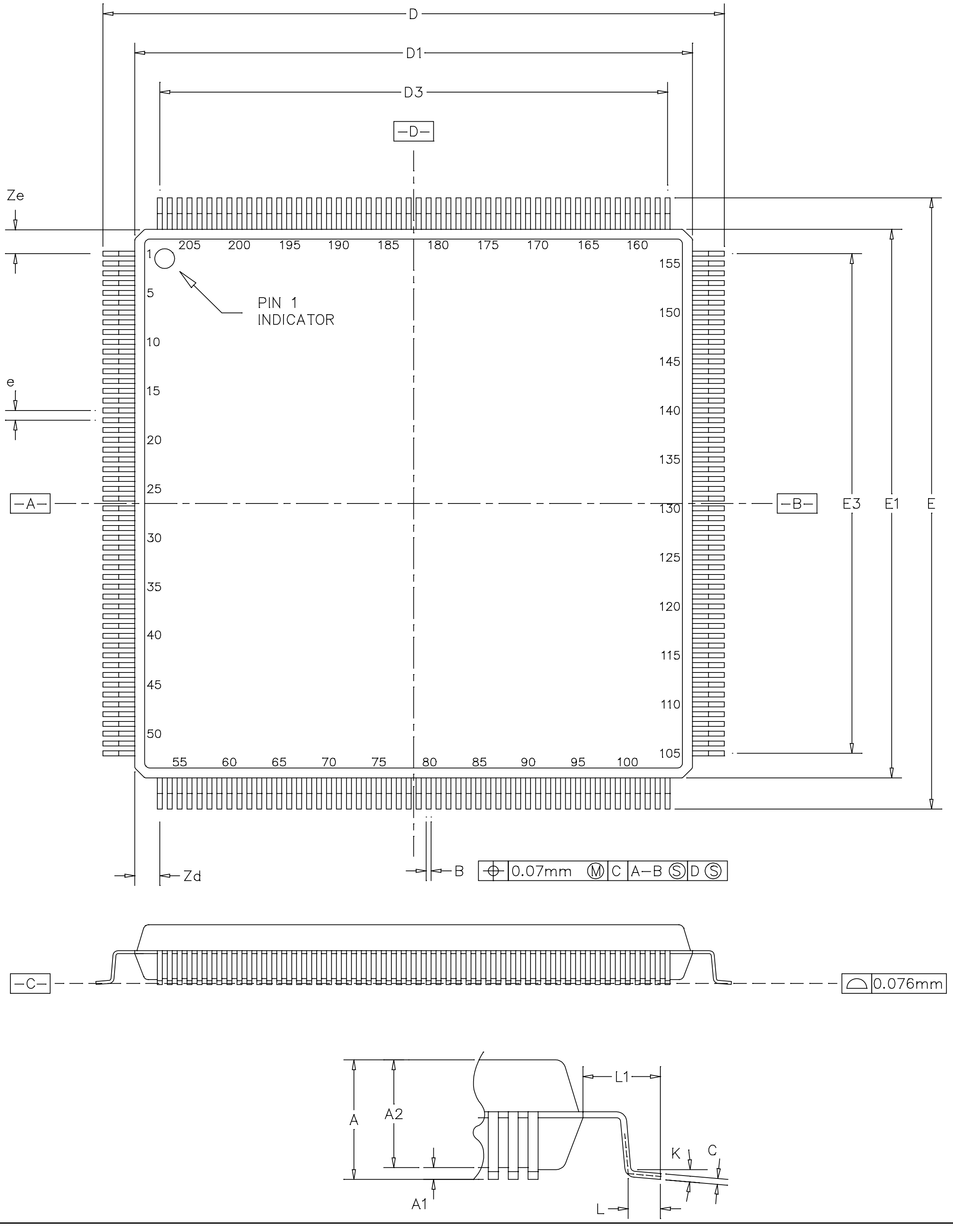


Figure 4.1 208 pin PQFP package dimensions

## 4.2 Pin list

| | | | | | |
|---|---|---|---|---|---|
| 1 | **Vdd** | P | 53 | **Vdda_0** | P |
| 2 | **Pio3_7** | I/O | 54 | **Vssa_0** | P |
| 3 | **Pio2_0/1** | I/O | 55 | **B_Out** | O |
| 4 | **Gnd** | P | 56 | **G_Out** | O |
| 5 | **Pio2_3** | I/O | 57 | **R_Out** | O |
| 6 | **Pio2_4** | I/O | 58 | **V_Ref_Dac_RGB** | I |
| 7 | **Pio2_5** | I/O | 59 | **I_Ref_Dac_RGB** | I |
| 8 | **Pio2_7** | I/O | 60 | **Vdda_1** | P |
| 9 | **Pio1_0/1** | I/O | 61 | **Vssa_1** | P |
| 10 | **Pio1_2** | I/O | 62 | **Y_Out** | O |
| 11 | **Pio1_5** | I/O | 63 | **C_Out** | O |
| 12 | **Pio1_6** | I/O | 64 | **CV_Out** | O |
| 13 | **Pio1_7** | I/O | 65 | **V_Ref_Dac_YCC** | I |
| 14 | **Pio4_7** | I/O | 66 | **I_Ref_Dac_YCC** | I |
| 15 | **Pio0_0/1** | I/O | 67 | **Vdd** | P |
| 16 | **Pio0_3** | I/O | 68 | **Gnd** | P |
| 17 | **Pio0_4** | I/O | 69 | **Ad4** | O |
| 18 | **Vdd** | P | 70 | **Ad5** | O |
| 19 | **Gnd** | P | 71 | **Ad6** | O |
| 20 | **Pio0_5** | I/O | 72 | **Ad7** | O |
| 21 | **Pio0_6** | I/O | 73 | **Ad8** | O |
| 22 | **Pio0_7** | I/O | 74 | **Ad9** | O |
| 23 | **Irq0** | I | 75 | **Vdd** | P |
| 24 | **Irq1** | I | 76 | **MemClkOut** | O |
| 25 | **Irq2** | I | 77 | **Gnd** | P |
| 26 | **Brm0 / Oslink_Sel** | I/O | 78 | **Ad0** | O |
| 27 | **Brm1 / BooFromRom** | I/O | 79 | **Ad1** | O |
| 28 | **Brm2** | O | 80 | **Ad2** | O |
| 29 | **not_Rst** | I | 81 | **Ad3** | O |
| 30 | **Sdav_Clk** | I/O | 82 | **Ad10** | O |
| 31 | **Sdav_Data** | I/O | 83 | **Ad11** | O |
| 32 | **Sdav_Dir** | I/O | 84 | **not_SdCS0** | O |
| 33 | **Osc_In / 27Mhz_Out** | I/O | 85 | **not_SdCS1** | O |
| 34 | **Vdd** | P | 86 | **Vdd** | P |
| 35 | **Gnd** | P | 87 | **Gnd** | P |
| 36 | **F_Data** | I | 88 | **not_SdRas** | O |
| 37 | **F_B_Clk** | I | 89 | **not_SdCas** | O |
| 38 | **F_P_Clk / D_Valid** | I | 90 | **not_SdWE** | O |
| 39 | **F_Error / P_Start** | I | 91 | **Dqml** | O |
| 40 | **Nrss_Clk** | O | 92 | **Dq0** | I/O |
| 41 | **Nrss_Out** | O | 93 | **Dq1** | I/O |
| 42 | **Nrss_In** | I | 94 | **Dq2** | I/O |
| 43 | **Pcm_ClkOut / A_C_Stb** | O | 95 | **Vdd** | P |
| 44 | **Pcm_Data / A_C_Data** | O | 96 | **Gnd** | P |
| 45 | **Pcm_ClkIn** | I/O | 97 | **Dq3** | I/O |
| 46 | **IrClk / A_Word_Clk** | O | 98 | **Dq4** | I/O |
| 47 | **A_C_Req** | I | 99 | **Dq5** | I/O |
| 48 | **A_Pts_Stb** | I | 100 | **Dq6** | I/O |
| 49 | **Vdd** | P | 101 | **Dq7** | I/1O |
| 50 | **Gnd** | P | 102 | **Vdd** | P |
| 51 | **not_Hsync** | I/O | 103 | **Gnd** | P |
| 52 | **Odd_not_Even** | I/O | 104 | **MemClkIn** | I |

| | | |
|---|---|---|
| 105 | **Dqmu** | O |
| 106 | **Dq8** | I/O |
| 107 | **Dq9** | I/O |
| 108 | **Dq10** | I/O |
| 109 | **Dq11** | I/O |
| 110 | **Vdd** | P |
| 111 | **Gnd** | P |
| 112 | **Dq12** | I/O |
| 113 | **Dq13** | I/O |
| 114 | **Dq14** | I/O |
| 115 | **Dq15** | I/O |
| 116 | **Link_Ext_Clk** | I |
| 117 | **Osd_Active** | I/O |
| 118 | **PixClk_27Mhz** | I |
| 119 | **Vdd** | P |
| 120 | **Gnd** | P |
| 121 | **not_WE0** | O |
| 122 | **not_WE1** | O |
| 123 | **not_OE** | O |
| 124 | **not_CE1** | O |
| 125 | **not_CE2** | O |
| 126 | **not_CE3** | O |
| 127 | **not_Ras0 / not_CE0** | O |
| 128 | **not_Ras1** | O |
| 129 | **not_Cas0** | O |
| 130 | **Vdd** | P |
| 131 | **Gnd** | P |
| 132 | **not_Cas1** | O |
| 133 | **ReadnotWrite** | O |
| 134 | **Vdd** | P |
| 135 | **Vdd** | P |
| 136 | **MemWait** | I |
| 137 | **ProcClockOut** | O |
| 138 | **Vdd** | P |
| 139 | **Vdd** | P |
| 140 | **Gnd** | P |
| 141 | **Data0** | I/O |
| 142 | **Data1** | I/O |
| 143 | **Data2** | I/O |
| 144 | **Data3** | I/O |
| 145 | **Data4** | I/O |
| 146 | **Data5** | I/O |
| 147 | **Data6** | I/O |
| 148 | **Data7** | I/O |
| 149 | **Vdd** | P |
| 150 | **Gnd** | P |
| 151 | **Data8** | I/O |
| 152 | **Data9** | I/O |
| 153 | **Data10** | I/O |
| 154 | **Data11** | I/O |
| 155 | **Data12** | I/O |
| 156 | **Data13** | I/O |
| 157 | **Data14** | I/O |
| 158 | **Data15** | I/O |
| 159 | **Vdd** | P |
| 160 | **Gnd** | P |
| 161 | **Adr1** | O |
| 162 | **Adr2** | O |
| 163 | **Adr3** | O |
| 164 | **Adr4** | O |
| 165 | **Adr5** | O |
| 166 | **Adr6** | O |
| 167 | **Adr7** | O |
| 168 | **Adr8** | O |
| 169 | **Adr9** | O |
| 170 | **Adr10** | O |
| 171 | **Vdd** | P |
| 172 | **Gnd** | P |
| 173 | **Adr11** | O |
| 174 | **Adr12** | O |
| 175 | **Adr13** | O |
| 176 | **Adr14** | O |
| 177 | **Adr15** | O |
| 178 | **Adr16** | O |
| 179 | **Adr17** | O |
| 180 | **Adr18** | O |
| 181 | **Adr19** | O |
| 182 | **Adr20** | O |
| 183 | **Adr21** | O |
| 184 | **Vdd** | P |
| 185 | **Gnd** | P |
| 186 | **Tdi** | I |
| 187 | **Tms** | I |
| 188 | **Tck** | I |
| 189 | **Tdo** | O |
| 190 | **not_Trst** | I |
| 191 | **Pio4_0** | I/O |
| 192 | **Pio4_1** | I/O |
| 193 | **Pio4_2** | I/O |
| 194 | **Pio4_3** | I/O |
| 195 | **Pio4_4** | I/O |
| 196 | **Pio4_5** | I/O |
| 197 | **Pio4_6** | I/O |
| 198 | **Pio1_3** | I/O |
| 199 | **Pio1_4** | I/O |
| 200 | **Gnd** | P |
| 201 | **Pio3_0** | I/O |
| 202 | **Pio3_1** | I/O |
| 203 | **Pio3_2** | I/O |
| 204 | **Pio3_3** | I/O |
| 205 | **Pio3_4** | I/O |
| 206 | **Pio3_5** | I/O |
| 207 | **Pio3_6** | I/O |
| 208 | **Vdd** | P |

# Part B The processor and memory

# 5 Central processing unit

The Central Processing Unit (CPU) is the ST20 32-bit processor core. It contains instruction processing logic, instruction and data pointers, and an operand register. It can directly access the high speed on-chip memory, which can store data or programs. Where larger amounts of memory are required, the processor can access memory via the External Memory Interface (EMI).

The processor provides high performance:

- Fast integer multiply - 4 cycle multiply
- Fast bit shift - single cycle barrel shifter
- Byte and part-word handling
- Scheduling and interrupt support
- 64-bit integer arithmetic support.

The scheduler provides a single level of pre-emption. In addition, multi-level pre-emption is provided by the interrupt subsystem, see Chapter 7 for details. Additionally, there is a per-priority trap handler to improve the support for arithmetic errors and illegal instructions.

## 5.1 Registers

The CPU contains six registers which are used in the execution of a sequential integer process. The six registers are:

- The workspace pointer (**Wptr**) which points to an area of store where local data is kept.
- The instruction pointer (**Iptr**) which points to the next instruction to be executed.
- The status register (**Status**).
- The **Areg**, **Breg** and **Creg** registers which form an evaluation stack.

The **Areg**, **Breg** and **Creg** registers are the sources and destinations for most arithmetic and logical operations. Loading a value into the stack pushes **Breg** into **Creg**, and **Areg** into **Breg**, before loading **Areg**. Storing a value from **Areg**, pops **Breg** into **Areg** and **Creg** into **Breg**. **Creg** is left undefined.

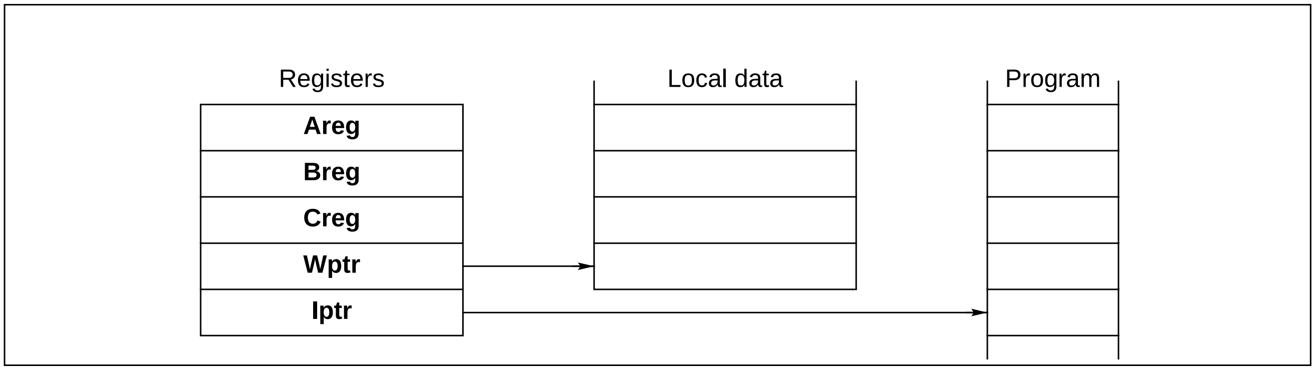


Figure 5.1 Registers used in sequential integer processes

Expressions are evaluated on the evaluation stack, and instructions refer to the stack implicitly. For example, the *add* instruction adds the top two values in the stack and places the result on the top of the stack. The use of a stack removes the need for instructions to explicitly specify the location of their operands. No hardware mechanism is provided to detect that more than three values have been loaded onto the stack; it is easy for the compiler to ensure that this never happens.

Note that a location in memory can be accessed relative to the workspace pointer, enabling the workspace to be of any size.

The use of shadow registers provides fast, simple and clean context switching.

## 5.2 Processes and concurrency

This section describes the default behavior of the CPU and it should be noted that the user can alter this behavior, for example by disabling timeslicing or installing a user scheduler.

A process starts, performs a number of actions, and then either stops without completing or terminates complete. Typically, a process is a sequence of instructions. The CPU can run several processes in parallel (concurrently). Processes may be assigned either high or low priority, and there may be any number of each.

The processor has a microcoded scheduler which enables any number of concurrent processes to be executed together, sharing the processor time. This removes the need for a software kernel, although kernels can still be written if desired.

At any time, a process may be

| | | |
|---|---|---|
| *active* | - | being executed, |
| | - | interrupted by a higher priority process, |
| | - | on a list waiting to be executed. |
| *inactive* | - | waiting to input, |
| | - | waiting to output, |
| | - | waiting until a specified time. |

The scheduler operates in such a way that inactive processes do not consume any processor time. Each active high priority process executes until it becomes inactive. The scheduler allocates a portion of the processor's time to each active low priority process in turn (see section 5.3). Active processes waiting to be executed are held in two linked lists of process work spaces, one of high priority processes and one of low priority processes. Each list is implemented using two registers, one of which points to the first process in the list, the other to the last. In the linked process list shown in Figure 5.2, process *S* is executing and *P*, *Q* and *R* are active, awaiting execution. Only the low priority process queue registers are shown; the high priority process ones behave in a similar manner.

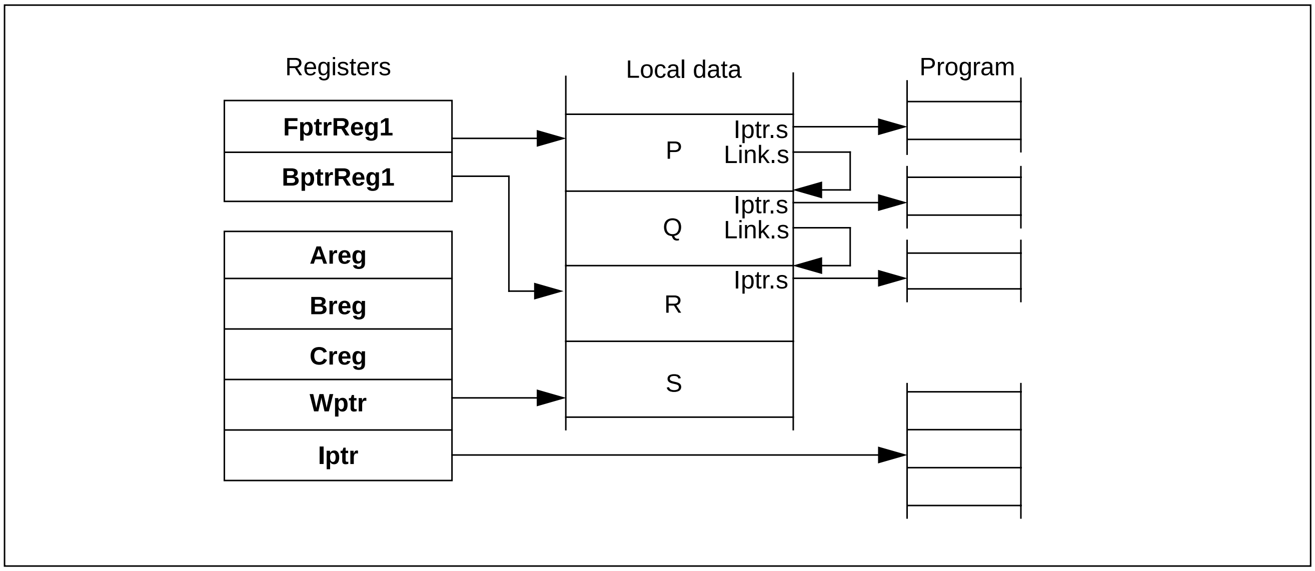


Figure 5.2 Linked process list

| Function | High priority | Low priority |
|---|---|---|
| Pointer to front of active process list | **FptrReg0** | **FptrReg1** |
| Pointer to back of active process list | **BptrReg0** | **BptrReg1** |

Table 5.1  Priority queue control registers

Each process runs until it has completed its action or is descheduled. In order for several processes to operate in parallel, a low priority process is only permitted to execute for a maximum of two timeslice periods. After this, the machine deschedules the current process at the next timeslicing point, adds it to the end of the low priority scheduling list and instead executes the next active process. The timeslice period is 1ms.

There are only certain instructions at which a process may be descheduled. These are known as descheduling points. A process may only be timesliced at certain descheduling points. These are known as timeslicing points and are defined in such a way that the operand stack is always empty. This removes the need for saving the operand stack when timeslicing. As a result, an expression evaluation can be guaranteed to execute without the process being timesliced part way through.

Whenever a process is unable to proceed, its instruction pointer is saved in the process workspace and the next process taken from the list.

The processor core provides a number of special instructions to support the process model, including *startp* (start process) and *endp* (end process). When a main process executes a parallel construct, *startp* is used to create the necessary additional concurrent processes. A *startp* instruction creates a new process by adding a new workspace to the end of the scheduling list, enabling the new concurrent process to be executed together with the ones already being executed. When a process is made active it is always added to the end of the list, and thus cannot pre-empt processes already on the same list.

The correct termination of a parallel construct is assured by use of the *endp* instruction. This uses a data structure that includes a counter of the parallel construct components which have still to terminate. The counter is initialized to the number of components before the processes are started. Each component ends with an *endp* instruction which decrements and tests the counter. For all but the last component, the counter is non zero and the component is descheduled. For the last component, the counter is zero and the main process continues.

## 5.3 Priority

The following section describes ‘default’ behavior of the CPU and it should be noted that the user can alter this behavior, for example, by disabling timeslicing and priority interrupts.

The processor can execute processes at one of two priority levels, one level for urgent (high priority) processes, one for less urgent (low priority) processes. A high priority process will always execute in preference to a low priority process if both are able to do so.

High priority processes are expected to execute for a short time. If one or more high priority processes are active, then the first on the queue is selected and executes until it has to wait for a communication, a timer input, or until it completes processing.

If no process at high priority is active, but one or more processes at low priority are active, then one is selected. Low priority processes are periodically timesliced to provide an even distribution of processor time between tasks which use a lot of computation.

If there are *n* low priority processes, then the maximum latency from the time at which a low priority process becomes active to the time when it starts processing is the order of 2*n* timeslice periods. It is then able to execute for between one and two timeslice periods, less any time taken by high priority processes. This assumes that no process monopolizes the time of the CPU; i.e. it has frequent timeslicing points.

The specific condition for a high priority process to start execution is that the CPU is idle or running at low priority and the high priority queue is non-empty.

If a high priority process becomes able to run while a low priority process is executing, the low priority process is temporarily stopped and the high priority process is executed. The state of the low priority process is saved into ‘shadow’ registers and the high priority process is executed. When no further high priority processes are able to run, the state of the interrupted low priority process is re-loaded from the shadow registers and the interrupted low priority process continues executing. Instructions are provided on the processor core to allow a high priority process to store the shadow registers to memory and to load them from memory. Instructions are also provided to allow a process to exchange an alternative process queue for either priority process queue (see ). These instructions allow extensions to be made to the scheduler for custom run-time kernels.

A low priority process may be interrupted after it has completed execution of any instruction. In addition, to minimize the time taken for an interrupting high priority process to start executing, the potentially time consuming instructions are interruptible. Also some instructions may be aborted, and are restarted when the process next becomes active (refer to the Instruction Set chapter).

## 5.4 Process communications

Communication between processes takes place over channels, and is implemented in hardware. Communication is point-to-point, synchronized and unbuffered. As a result, a channel needs no process queue, no message queue and no message buffer.

A channel between two processes executing on the same CPU is implemented by a single word in memory; a channel between processes executing on different processors is implemented by point-to-point links. The processor provides a number of operations to support message passing, the most important being *in* (input message) and *out* (output message).

The *in* and *out* instructions use the address of the channel to determine whether the channel is internal or external. This means that the same instruction sequence can be used for both hard and soft channels, allowing a process to be written and compiled without knowledge of where its channels are implemented.

Communication takes place when both the inputting and outputting processes are ready. Consequently, the process which first becomes ready must wait until the second one is also ready. The inputting and outputting processes only become active when the communication has completed.

A process performs an input or output by loading the evaluation stack with, a pointer to a message, the address of a channel, and a count of the number of bytes to be transferred, and then executing an *in* or *out* instruction.

## 5.5 Timers

There are two 32-bit hardware timer clocks which ‘tick’ periodically. These are independent of any on-chip peripheral real time clock. The timers provide accurate process timing, allowing processes to deschedule themselves until a specific time.

One timer is accessible only to high priority processes and is incremented approximately every microsecond, cycling completely in approximately 4295 seconds. The other is accessible only to low priority processes and runs 64 times slower, giving 15625 ticks per second. It has a full period of approximately 76 hours.

Actual timer speeds are derived from the processor speed **ProcClockOut** and are given in the *Clocks* chapter. The periods may be calculated as follows:

$$\mathit{High\_priority\_clock\_period} = 1\mu\mathrm{s} \times \mathit{Nominal\_speed} \,/\, \textbf{ProcClockOut}\_\mathit{speed}$$

$$\mathit{Low\_priority\_clock\_period} = \mathit{High\_priority\_clock\_period} \times 64$$

| Register | Function |
|---|---|
| **ClockReg0** | Current value of high priority (level 0) process clock. |
| **ClockReg1** | Current value of low priority (level 1) process clock. |
| **TnextReg0** | Indicates time of earliest event on high priority (level 0) timer queue. |
| **TnextReg1** | Indicates time of earliest event on low priority (level 1) timer queue. |
| **TptrReg0** | High priority timer queue. |
| **TptrReg1** | Low priority timer queue. |

Table 5.2 Timer registers

The current value of the processor clock can be read by executing a *ldtimer* (load timer) instruction. A process can arrange to perform a *tin* (timer input), in which case it will become ready to execute after a specified time has been reached. The *tin* instruction requires a time to be specified. If this time is in the 'past' then the instruction has no effect. If the time is in the 'future' then the process is descheduled. When the specified time is reached the process becomes active. In addition, the *ldclock* (load clock), *stclock* (store clock) instructions allow total control over the clock value and the *clockenb* (clock enable), *clockdis* (clock disable) instructions allow each clock to be individually stopped and re-started.

Figure 5.3 shows two processes waiting on the timer queue, one waiting for time 21, the other for time 31.

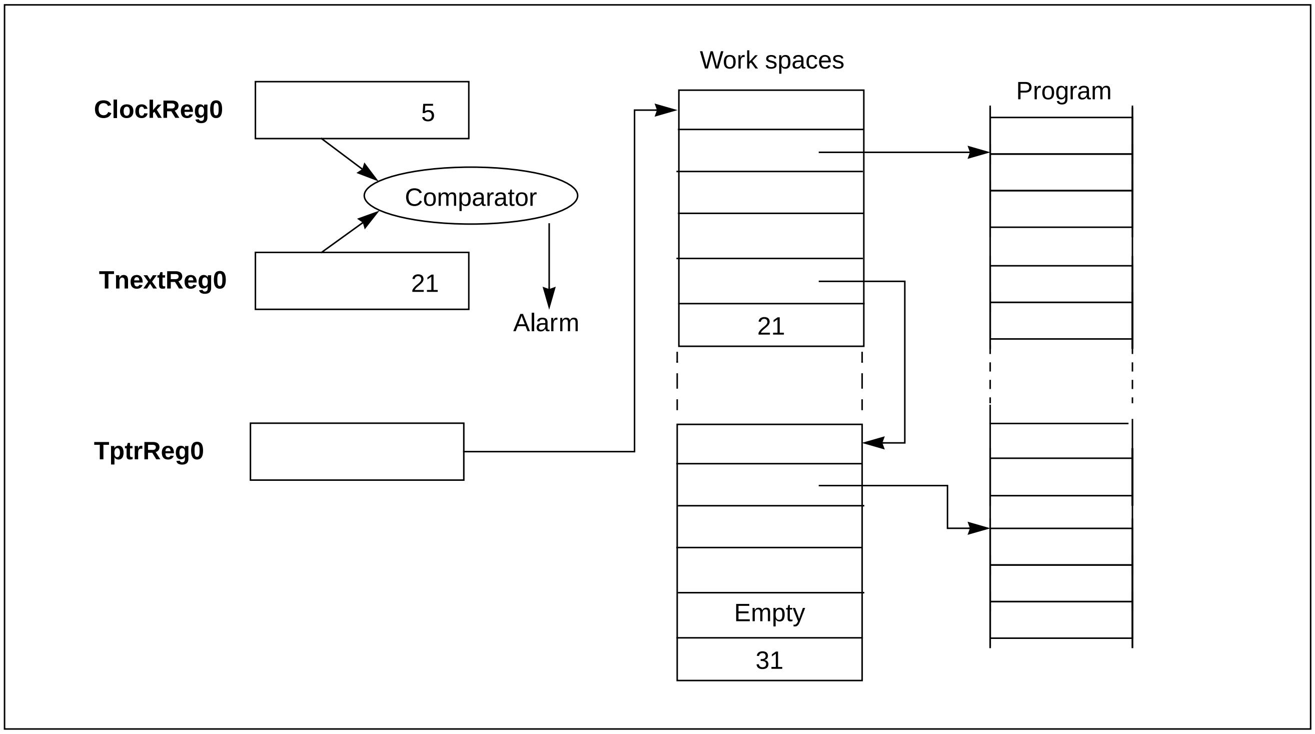


Figure 5.3 Timer registers

## 5.6 Traps and exceptions

A software error, such as arithmetic overflow or array bounds violation, can cause an error flag to be set in the CPU. The flag is directly connected to the **ErrorOut** pin. Both the flag and the pin can be ignored, or the CPU stopped. Stopping the CPU on an error means that the error cannot cause further corruption. As well as containing the error in this way it is possible to determine the state of the CPU and its memory at the time the error occurred. This is particularly useful for postmortem debugging where the debugger can be used to examine the state and history of the processor leading up to and causing the error condition.

In addition, if a trap handler process is installed, a variety of traps/exceptions can be trapped and handled by software. A user supplied trap handler routine can be provided for each high/low process priority level. The handler is started when a trap occurs and is given the reason for the trap.

The trap handler is not re-entrant and must not cause a trap itself within the same group. All traps can be individually masked.

### 5.6.1 Trap groups

The trap mechanism is arranged on a per priority basis. For each priority there is a handler for each group of traps, as shown in Figure 5.4.

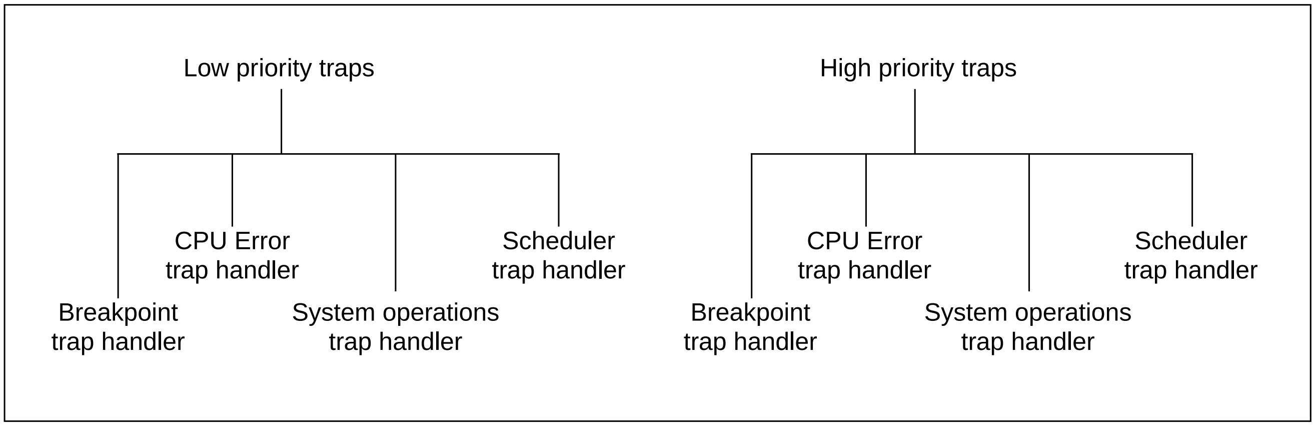


Figure 5.4 Trap arrangement

There are four groups of traps, as detailed below.

- Breakpoint

  This group consists of the *Breakpoint* trap. The breakpoint instruction (*j0*) calls the breakpoint routine via the trap mechanism.

- Errors

  The traps in this group are *IntegerError* and *Overflow*. *Overflow* represents arithmetic overflow, such as arithmetic results which do not fit in the result word. *IntegerError* represents errors caused when data is erroneous, for example when a range checking instruction finds that data is out of range.

- System operations

  This group consists of the *LoadTrap*, *StoreTrap* and *IllegalOpcode* traps. The *IllegalOpcode* trap is signalled when an attempt is made to execute an illegal instruction. The *LoadTrap* and *StoreTrap* traps allow a kernel to intercept attempts by a monitored process to change or examine trap handlers or trapped process information. It enables a user program to signal to a kernel that it wishes to install a new trap handler.

- Scheduler

  The scheduler trap group consists of the *ExternalChannel, InternalChannel, Timer, TimeSlice, Run, Signal, ProcessInterrupt* and *QueueEmpty* traps. The *ProcessInterrupt* trap signals that the machine has performed a priority interrupt from low to high. The *QueueEmpty* trap indicates that there is no further executable work to perform. The other traps in this group indicate that the hardware scheduler wants to schedule a process on a process queue, with the different traps enabling the different sources of this to be monitored.

The scheduler traps enable a software scheduler kernel to use the hardware scheduler to implement a multi-priority software scheduler.

Note that scheduler traps are different from other traps as they are caused by the microscheduler rather than by an executing process.

Trap groups encoding is shown in Table 5.3 below. These codes are used to identify trap groups to various instructions.

| **Trap group** | **Code** |
|---|---|
| Breakpoint | 0 |
| CPU errors | 1 |
| System operations | 2 |
| Scheduler | 3 |

Table 5.3  Trap group codes

In addition to the trap groups mentioned above, the **CauseError** flag in the **Status** register is used to signal when a trap condition has been activated by the *causeerror* instruction. It can be used to indicate when trap conditions have occurred due to the user setting them, rather than by the system.

### 5.6.2 Events that can cause traps

Table 5.4 summarizes the events that can cause traps and gives the encoding of bits in the trap **Status** and **Enable** words.

| Trap cause | Status/Enable codes | Trap group | Comments |
|---|---|---|---|
| *Breakpoint* | 0 | 0 | When a process executes the breakpoint instruction (*j0*) then it traps to its trap handler. |
| *IntegerError* | 1 | 1 | Integer error other than integer overflow - e.g. explicitly checked or explicitly set error. |
| *Overflow* | 2 | 1 | Integer overflow or integer division by zero. |
| *IllegalOpcode* | 3 | 2 | Attempt to execute an illegal instruction. This is signalled when *opr* is executed with an invalid operand. |
| *LoadTrap* | 4 | 2 | When the trap descriptor is read with the *ldtraph* instruction or when the trapped process status is read with the *ldtrapped* instruction. |
| *StoreTrap* | 5 | 2 | When the trap descriptor is written with the *sttraph* instruction or when the trapped process status is written with the *sttrapped* instruction. |
| *InternalChannel* | 6 | 3 | Scheduler trap from internal channel. |
| *ExternalChannel* | 7 | 3 | Scheduler trap from external channel. |
| *Timer* | 8 | 3 | Scheduler trap from timer alarm. |
| *Timeslice* | 9 | 3 | Scheduler trap from timeslice. |
| *Run* | 10 | 3 | Scheduler trap from *runp* (run process) or *startp* (start process). |
| *Signal* | 11 | 3 | Scheduler trap from *signal.* |
| *ProcessInterrupt* | 12 | 3 | Start executing a process at a new priority level. |
| *QueueEmpty* | 13 | 3 | Caused by no process active at a priority level. |
| *CauseError* | 15 (Status only) | Any, encoded 0-3 | Signals that the *causeerror* instruction set the trap flag. |

Table 5.4 Trap causes and **Status**/**Enable** codes

### 5.6.3 Trap handlers

For each trap handler there is a trap handler structure and a trapped process structure. Both the trap handler structure and the trapped process structure are in memory and can be accessed via instructions, see section 5.6.4.

The trap handler structure specifies what should happen when a trap condition is present, see Table 5.5.

The trapped process structure saves some of the state of the process that was running when the trap was taken, see Table 5.6.

In addition, for each priority, there is an **Enables** register and a **Status** register. The **Enables** register contains flags to enable each cause of trap. The **Status** register contains flags to indicate which trap conditions have been detected. The **Enables** and **Status** register bit encodings are given in Table 5.4.

| | **Comments** | **Location** |
|---|---|---|
| **Iptr** | **Iptr** of trap handler process. | Base + 3 |
| **Wptr** | **Wptr** of trap handler process. A null **Wptr** indicates that a trap handler has not been installed. | Base + 2 |
| **Status** | Contains the **Status** register that the trap handler starts with. | Base + 1 |
| **Enables** | A word which encodes the trap enable and global interrupt masks, which will be ANDed with the existing masks to allow the trap handler to disable various events while it runs. | Base + 0 |

Table 5.5 Trap handler structure

| | **Comments** | **Location** |
|---|---|---|
| **Iptr** | Points to the instruction after the one that caused the trap condition. | Base + 3 |
| **Wptr** | **Wptr** of the process that was running when the trap was taken. | Base + 2 |
| **Status** | The relevant trap bit is set, see Table 5.3 for trap codes. | Base + 1 |
| **Enables** | Interrupt enables. | Base + 0 |

Table 5.6 Trapped process structure

A trap will be taken at an interruptible point if a trap is set and the corresponding trap enable bit is set in the **Enables** register. If the trap is not enabled then nothing is done with the trap condition. If the trap is enabled then the corresponding bit is set in the **Status** register to indicate the trap condition has occurred.

When a process takes a trap the processor saves the existing **Iptr**, **Wptr**, **Status** and **Enables** in the trapped process structure. It then loads **Iptr**, **Wptr** and **Status** from the equivalent trap handler structure and ANDs the value in **Enables** with the value in the structure. This allows the user to disable various events while in the handler, in particular a trap handler must disable all the traps of its trap group to avoid the possibility of a handler trapping to itself.

The trap handler then executes. The values in the trapped process structure can be examined using the *ldtrapped* instruction (see section 5.6.4). When the trap handler has completed its operation it returns to the trapped process via the *tret* (trap return) instruction. This reloads the values saved in the trapped process structure and clears the trap flag in **Status**.

Note that when a trap handler is started, **Areg**, **Breg** and **Creg** are not saved. The trap handler must save the **Areg**, **Breg**, **Creg** registers using *stl* (store local).

### 5.6.4 Trap instructions

Trap handlers and trapped processes can be set up and examined via the *ldtraph*, *sttraph*, *ldtrapped* and *sttrapped* instructions. Table 5.7 describes the instructions that may be used when dealing with traps.

| Instruction | Meaning | Use |
|---|---|---|
| *ldtraph* | load trap handler | Load the trap handler from memory to the trap handler descriptor. |
| *sttraph* | store trap handler | Store an existing trap handler descriptor to memory. |
| *ldtrapped* | load trapped | Load replacement trapped process status from memory. |
| *sttrapped* | store trapped | Store trapped process status to memory. |
| *trapenb* | trap enable | Enable traps. |
| *trapdis* | trap disable | Disable traps. |
| *tret* | trap return | Used to return from a trap handler. |
| *causeerror* | cause error | Program can simulate the occurrence of an error. |

Table 5.7 Instructions which may be used when dealing with traps

The first four instructions transfer data to/from the trap handler structures or trapped process structures from/to an area in memory. In these instructions **Areg** contains the trap group code (see Table 5.3) and **Breg** points to the 4 word area of memory used as the source or destination of the transfer. In addition **Creg** contains the priority of the handler to be installed/examined in the case of *ldtraph* or *sttraph. ldtrapped* and *sttrapped* apply only to the current priority.

If the *LoadTrap* trap is enabled then *ldtraph* and *ldtrapped* do not perform the transfer but set the **LoadTrap** trap flag. If the *StoreTrap* trap is enabled then *sttraph* and *sttrapped* do not perform the transfer but set the **StoreTrap** trap flag.

The trap enable masks are encoded by an array of bits (see Table 5.4) which are set to indicate which traps are enabled. This array of bits is stored in the lower half-word of the **Enables** register. There is an **Enables** register for each priority. Traps are enabled or disabled by loading a mask into **Areg** with bits set to indicate which traps are to be affected and the priority to affect in **Breg**. Executing *trapenb* ORs the mask supplied in **Areg** with the trap enables mask in the **Enables** register for the priority in **Breg**. Executing *trapdis* negates the mask supplied in **Areg** and ANDs it with the trap enables mask in the **Enables** register for the priority in **Breg**. Both instructions return the previous value of the trap enables mask in **Areg**.

### 5.6.5 Restrictions on trap handlers

There are various restrictions that must be placed on trap handlers to ensure that they work correctly.

1. Trap handlers must not deschedule or timeslice. Trap handlers alter the **Enables** masks, therefore they must not allow other processes to execute until they have completed.
2. Trap handlers must have their **Enable** masks set to mask all traps in their trap group to avoid the possibility of a trap handler trapping to itself.
3. Trap handlers must terminate via the *tret* (trap return) instruction. The only exception to this is that a scheduler kernel may use *restart* to return to a previously shadowed process.

# 6 Instruction set

This chapter provides information on the ST20-C2 instruction set. It contains tables listing all the instructions, and where applicable provides details of the number of processor cycles taken by an instruction.

The instruction set has been designed for simple and efficient compilation of high-level languages. All instructions have the same format, designed to give a compact representation of the operations occurring most frequently in programs.

Each instruction consists of a single byte divided into two 4-bit parts. The four most significant bits (MSB) of the byte are a function code and the four least significant bits (LSB) are a data value, as shown in Figure 6.1.

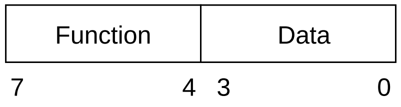


Figure 6.1 Instruction format

For further information on the instruction set refer to the *ST20C2 Instruction Set Manual* (document number 72-TRN-273).

## 6.1 Instruction cycles

Timing information is available for some instructions. However, it should be noted that many instructions have ranges of timings which are data dependent.

Where included, timing information is based on the number of clock cycles assuming any memory accesses are to 2 cycle internal memory and no other subsystem is using memory. Actual time will be dependent on the speed of external memory and memory bus availability.

Note that the actual time can be increased by:

1. the instruction requiring a value on the register stack from the final memory read in the previous instruction – the current instruction will stall until the value becomes available.
2. the first memory operation in the current instruction can be delayed while a preceding memory operation completes - any two memory operations can be in progress at any time, any further operation will stall until the first completes.
3. memory operations in current instructions can be delayed by access by instruction fetch or subsystems to the memory interface.
4. there can be a delay between instructions while the instruction fetch unit fetches and partially decodes the next instruction – this will be the case whenever an instruction causes the instruction flow to jump.

Note that the instruction timings given refer to ‘standard’ behavior and may be different if, for example, traps are set by the instruction.

## 6.2 Instruction characteristics

Table 6.2 gives the basic function code of each of the primary instructions. Where the operand is less than 16, a single byte encodes the complete instruction. If the operand is greater than 15, one prefix instruction (*pfix*) is required for each additional four bits of the operand. If the operand is negative the first prefix instruction will be *nfix*. Examples of *pfix* and *nfix* coding are given in Table 6.1.

| Mnemonic | | Function code | Memory code |
|---|---|---|---|
| *ldc* | #3 | #4 | #43 |
| *ldc* | #35 | | |
| is coded as | | | |
| *pfix* | #3 | #2 | #23 |
| *ldc* | #5 | #4 | #45 |
| *ldc* | #987 | | |
| is coded as | | | |
| *pfix* | #9 | #2 | #29 |
| *pfix* | #8 | #2 | #28 |
| *ldc* | #7 | #4 | #47 |
| *ldc* | -31 (*ldc* #FFFFFFE1) | | |
| is coded as | | | |
| *nfix* | #1 | #6 | #61 |
| *ldc* | #1 | #4 | #41 |

Table 6.1 Prefix coding

Any instruction which is not in the instruction set tables is an invalid instruction and is flagged illegal, returning an error code to the trap handler, if loaded and enabled.

The **Notes** column of the tables indicates the features of an instruction as described in Table 6.2.

| Ident | Feature |
|---|---|
| E | Instruction can set an *IntegerError* trap |
| L | Instruction can cause a *LoadTrap* trap |
| S | Instruction can cause a *StoreTrap* trap |
| O | Instruction can cause an *Overflow* trap |
| I | Interruptible instruction |
| A | Instruction can be aborted and later restarted. |
| D | Instruction can deschedule |
| T | Instruction can timeslice |

Table 6.2 Instruction features

## 6.3 Instruction set tables

| Function code | Memory code | Mnemonic | Processor cycles | Name | Notes |
|---|---|---|---|---|---|
| 0 | 0X | j | 5 | jump | D, T |
| 1 | 1X | ldlp | 1 | load local pointer | |
| 2 | 2X | pfix | 0 to 1 | prefix | |
| 3 | 3X | ldnl | 2 | load non-local | |
| 4 | 4X | ldc | 1 | load constant | |
| 5 | 5X | ldnlp | 1 | load non-local pointer | |
| 6 | 6X | nfix | 0 to 1 | negative prefix | |
| 7 | 7X | ldl | 1 | load local | |
| 8 | 8X | adc | 1 | add constant | O |
| 9 | 9X | call | 8 | call | |
| A | AX | cj | 1 or 5 | conditional jump | |
| B | BX | ajw | 2 | adjust workspace | |
| C | CX | eqc | 1 | equals constant | |
| D | DX | stl | 1 | store local | |
| E | EX | stnl | 2 | store non-local | |
| F | FX | opr | 0 | operate | |

Table 6.2  Primary functions

| Memory code | Mnemonic | Processor cycles | Name | Notes |
|---|---|---|---|---|
| 22FA | testpranal | 2 | test processor analyzing | |
| 23FE | saveh | 3 | save high priority queue registers | |
| 23FD | savel | 3 | save low priority queue registers | |
| 21F8 | sthf | 1 | store high priority front pointer | |
| 25F0 | sthb | 1 | store high priority back pointer | |
| 21FC | stlf | 1 | store low priority front pointer | |
| 21F7 | stlb | 1 | store low priority back pointer | |
| 25F4 | sttimer | 2 | store timer | |
| 2127FC | lddevid | 1 | load device identity | |
| 27FE | ldmemstartval | 1 | load value of **MemStart** address | |

Table 6.3  Processor initialization operation codes

| Memory code | Mnemonic | Processor cycles | Name | Notes |
|---|---|---|---|---|
| 24F6 | and | 1 | and | |
| 24FB | or | 1 | or | |
| 23F3 | xor | 1 | exclusive or | |
| 23F2 | not | 1 | bitwise not | |
| 24F1 | shl | 1 | shift left | |
| 24F0 | shr | 1 | shift right | |
| | | | | |
| F5 | add | 1 | add | A, O |
| FC | sub | 1 | subtract | A, O |
| 25F3 | mul | 4 | multiply | A, O |
| 27F2 | fmul | 6 | fractional multiply | A, O |
| 22FC | div | 5 to 37 | divide | A, O |
| 21FF | rem | 5 to 40 | remainder | A, O |
| F9 | gt | 1 | greater than | A |
| 25FF | gtu | 1 | greater than unsigned | A |
| F4 | diff | 1 | difference | |
| 25F2 | sum | 1 | sum | |
| F8 | prod | 4 | product | A |
| 26F8 | satadd | 2 | saturating add | A |
| 26F9 | satsub | 2 | saturating subtract | A |
| 26FA | satmul | 5 | saturating multiply | A |

Table 6.4 Arithmetic/logical operation codes

| Memory code | Mnemonic | Processor cycles | Name | Notes |
|---|---|---|---|---|
| 21F6 | ladd | 2 | long add | A, O |
| 23F8 | lsub | 2 | long subtract | A, O |
| 23F7 | lsum | 2 | long sum | |
| 24FF | ldiff | 2 | long diff | |
| 23F1 | lmul | 5 to 6 | long multiply | A |
| 21FA | ldiv | 5 to 39 | long divide | A, O |
| 23F6 | lshl | 2 | long shift left | A |
| 23F5 | lshr | 2 | long shift right | A |
| 21F9 | norm | 2 to 5 | normalize | A |
| 26F4 | slmul | 5 | signed long multiply | A, O |
| 26F5 | sulmul | 5 | signed times unsigned long multiply | A, O |

Table 6.5  Long arithmetic operation codes

| Memory code | Mnemonic | Processor cycles | Name | Notes |
|---|---|---|---|---|
| F0 | rev | 1 | reverse | |
| 23FA | xword | 4 | extend to word | A |
| 25F6 | cword | 3 | check word | A, E |
| 21FD | xdble | 2 | extend to double | |
| 24FC | csngl | 3 | check single | A, E |
| 24F2 | mint | 1 | minimum integer | |
| 25FA | dup | 1 | duplicate top of stack | |
| 27F9 | pop | 1 | pop processor stack | |
| 68FD | reboot | 1 | reboot | |

Table 6.6  General operation codes

| Memory code | Mnemonic | Processor cycles | Name | Notes |
|---|---|---|---|---|
| F2 | bsub | 1 | byte subscript | |
| FA | wsub | 1 | word subscript | |
| 28F1 | wsubdb | 1 | form double word subscript | |
| 23F4 | bcnt | 1 | byte count | |
| 23FF | wcnt | 1 | word count | |
| F1 | lb | 1 | load byte | |
| 23FB | sb | 2 | store byte | |
| | | | | |
| 24FA | move | | move message | I |

Table 6.7 Indexing/array operation codes

| Memory code | Mnemonic | Processor cycles | Name | Notes |
|---|---|---|---|---|
| 22F2 | ldtimer | 1 | load timer | |
| 22FB | tin | | timer input | I |
| 24FE | talt | 3 | timer alt start | |
| 25F1 | taltwt | | timer alt wait | D, I |
| 24F7 | enbt | 2 to 8 | enable timer | |
| 22FE | dist | | disable timer | I |

Table 6.8 Timer handling operation codes

| Memory code | Mnemonic | Processor cycles | Name | Notes |
|---|---|---|---|---|
| F7 | in | | input message | D |
| FB | out | | output message | D |
| FF | outword | | output word | D |
| FE | outbyte | | output byte | D |
| | | | | |
| 24F3 | alt | 2 | alt start | |
| 24F4 | altwt | 4 to 7 | alt wait | D |
| 24F5 | altend | 9 | alt end | |
| | | | | |
| 24F9 | enbs | 1 to 2 | enable skip | |
| 23F0 | diss | 1 | disable skip | |
| | | | | |
| 21F2 | resetch | 3 | reset channel | |
| 24F8 | enbc | 2 to 5 | enable channel | |
| 22FF | disc | 2 to 7 | disable channel | |

Table 6.9  Input and output operation codes

| Memory code | Mnemonic | Processor cycles | Name | Notes |
|---|---|---|---|---|
| 22F0 | ret | 3 | return | |
| 21FB | ldpi | 1 | load pointer to instruction | |
| 23FC | gajw | 3 | general adjust workspace | |
| F6 | gcall | 6 | general call | |
| 22F1 | lend | 5 to 8 | loop end | T |

Table 6.10  Control operation codes

| Memory code | Mnemonic | Processor cycles | Name | Notes |
|---|---|---|---|---|
| FD | startp | 5 | start process | |
| F3 | endp | 4 to 6 | end process | D |
| 23F9 | runp | 3 | run process | |
| 21F5 | stopp | 2 | stop process | |
| 21FE | ldpri | 1 | load current priority | |

Table 6.11  Scheduling operation codes

| Memory code | Mnemonic | Processor cycles | Name | Notes |
|---|---|---|---|---|
| 21F3 | csub0 | 2 | check subscript from 0 | A, E |
| 24FD | ccnt1 | 3 | check count from 1 | A, E |
| 22F9 | testerr | 2 | test error false and clear | |
| 21F0 | seterr | 2 | set error | |
| 25F5 | stoperr | 2 to 3 | stop on error (no error) | D |
| 25F7 | clrhalterr | 1 | clear halt-on-error | |
| 25F8 | sethalterr | 1 | set halt-on-error | |
| 25F9 | testhalterr | 2 | test halt-on-error | |

Table 6.12 Error handling operation codes

| Memory code | Mnemonic | Processor cycles | Name | Notes |
|---|---|---|---|---|
| 25FB | move2dinit | 3 | initialize data for 2D block move | |
| 25FC | move2dall | | 2D block copy | I |
| 25FD | move2dnonzero | | 2D block copy non-zero bytes | I |
| 25FE | move2dzero | | 2D block copy zero bytes | I |

Table 6.13 2D block move operation codes

| Memory code | Mnemonic | Processor cycles | Name | Notes |
|---|---|---|---|---|
| 27F4 | crcword | 36 | calculate crc on word | A |
| 27F5 | crcbyte | 12 | calculate crc on byte | A |
| 27F6 | bitcnt | 3 | count bits set in word | A |
| 27F7 | bitrevword | 2 | reverse bits in word | |
| 27F8 | bitrevnbits | 2 | reverse bottom n bits in word | A |

Table 6.14 CRC and bit operation codes

| Memory code | Mnemonic | Processor cycles | Name | Notes |
|---|---|---|---|---|
| 27F3 | cflerr | 3 | check floating point error | E |
| 29FC | fptesterr | 1 | load value true (FPU not present) | |
| 26F3 | unpacksn | 10 | unpack single length floating point number | A |
| 26FD | roundsn | 7 | round single length floating point number | A |
| 26FC | postnormsn | 9 | post-normalize correction of single length floating point number | A |
| 27F1 | ldinf | 1 | load single length infinity | |

Table 6.15 Floating point support operation codes

| Memory code | Mnemonic | Processor cycles | Name | Notes |
|---|---|---|---|---|
| 2CF7 | cir | 3 | check in range | A, E |
| 2CFC | ciru | 3 | check in range unsigned | A, E |
| 2BFA | cb | 3 | check byte | A, E |
| 2BFB | cbu | 2 | check byte unsigned | A, E |
| 2FFA | cs | 3 | check sixteen | A, E |
| 2FFB | csu | 2 | check sixteen unsigned | A, E |
| 2FF8 | xsword | 3 | sign extend sixteen to word | A |
| 2BF8 | xbword | 3 | sign extend byte to word | A |

Table 6.16 Range checking and conversion instructions

| Memory code | Mnemonic | Processor cycles | Name | Notes |
|---|---|---|---|---|
| 2CF1 | ssub | 1 | sixteen subscript | |
| 2CFA | ls | 1 | load sixteen | |
| 2CF8 | ss | 2 | store sixteen | |
| 2BF9 | lbx | 1 | load byte and sign extend | |
| 2FF9 | lsx | 1 | load sixteen and sign extend | |

Table 6.17 Indexing/array instructions

| Memory code | Mnemonic | Processor cycles | Name | Notes |
|---|---|---|---|---|
| 2FF0 | devlb | 3 | device load byte | A |
| 2FF2 | devls | 3 | device load sixteen | A |
| 2FF4 | devlw | 3 | device load word | A |
| 62F4 | devmove |  | device move | I |
| 2FF1 | devsb | 3 | device store byte | A |
| 2FF3 | devss | 3 | device store sixteen | A |
| 2FF5 | devsw | 3 | device store word | A |

Table 6.18  Device access instructions

| Memory code | Mnemonic | Processor cycles | Name | Notes |
|---|---|---|---|---|
| 60F5 | wait | 5 to 11 | wait | D |
| 60F4 | signal | 7 to 12 | signal |  |

Table 6.19  Semaphore instructions

| Memory code | Mnemonic | Processor cycles | Name | Notes |
|---|---|---|---|---|
| 60F0 | swapqueue | 4 | swap scheduler queue |  |
| 60F1 | swaptimer | 5 | swap timer queue |  |
| 60F2 | insertqueue | 3 to 4 | insert at front of scheduler queue |  |
| 60F3 | timeslice | 3 to 4 | timeslice |  |
| 60FC | ldshadow | 6 to 31 | load shadow registers | A |
| 60FD | stshadow | 6 to 17 | store shadow registers | A |
| 62FE | restart | 20 | restart |  |
| 62FF | causeerror | 7 to 8 | cause error |  |
| 61FF | iret | 3 to 11 | interrupt return |  |
| 2BF0 | settimeslice | 2 | set timeslicing status |  |
| 2CF4 | intdis | 2 | interrupt disable |  |
| 2CF5 | intenb | 2 | interrupt enable |  |
| 2CFD | gintdis | 5 | global interrupt disable |  |
| 2CFE | gintenb | 5 | global interrupt enable |  |

Table 6.20  Scheduling support instructions

| Memory code | Mnemonic | Processor cycles | Name | Notes |
|---|---|---|---|---|
| 26FE | ldtraph | 12 | load trap handler | L |
| 2CF6 | ldtrapped | 12 | load trapped process status | L |
| 2CFB | sttrapped | 12 | store trapped process status | S |
| 26FF | sttraph | 12 | store trap handler | S |
| 60F7 | trapenb | 4 | trap enable | |
| 60F6 | trapdis | 4 | trap disable | |
| 60FB | tret | 8 to 10 | trap return | |

Table 6.21 Trap handler instructions

| Memory code | Mnemonic | Processor cycles | Name | Notes |
|---|---|---|---|---|
| 68FC | ldprodid | 1 | load product identity | |
| 63F0 | nop | 1 | no operation | |

Table 6.22 Processor initialization and no operation instructions

| Memory code | Mnemonic | Processor cycles | Name | Notes |
|---|---|---|---|---|
| 64FF | clockenb | 2 | clock enable | |
| 64FE | clockdis | 2 | clock disable | |
| 64FD | ldclock | 2 | load clock | |
| 64FC | stclock | 2 | store clock | |

Table 6.23 Clock instructions

# 7 Interrupt controller

The interrupt system allows an on-chip subsystem or external interrupt pin to interrupt the currently running process in order to run an interrupt handling process.

Depending on the device, an interrupt may be signalled by one of the following:

- a signal on an external **Irq** pin,
- a signal from an internal peripheral or subsystem,
- software asserting an interrupt in the **Pending** register.

Interrupts are implemented using an on-chip interrupt controller peripheral and an on-chip interrupt level controller. The interrupt level controller (described in section 7.6) multiplexes incoming interrupts onto the eight programmable interrupt inputs of the interrupt controller. This multiplexing is controllable by software.

The interrupt controller supports eight prioritized interrupts as inputs, and manages the pending interrupts. This allows nested pre-emptive interrupts for real-time system design.

All interrupts are at a higher priority than the low priority process queue. Each interrupt can be programmed to be at a lower or higher priority than the high priority process queue, by writing to the priority bit in the **HandlerWptr** registers. Interrupts which are specified as higher priority must be contiguous from the highest numbered interrupt downwards. For example, if 4 interrupts are programmed as higher priority and 4 as lower priority the higher priority interrupts must be **Interrupt7:4** and the lower priority interrupts **Interrupt3:0**.

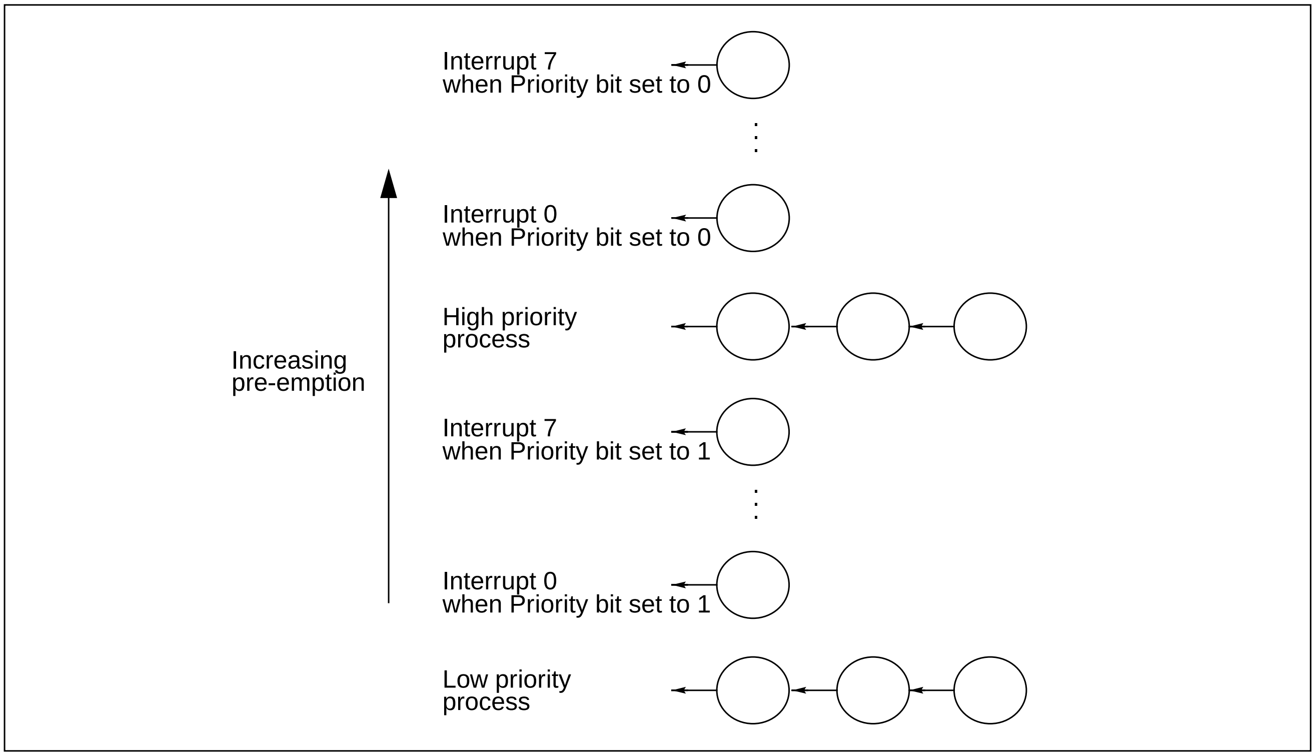

Interrupt 7
when Priority bit set to 0
Interrupt 0
when Priority bit set to 0
High priority
process
Increasing
pre-emption
Interrupt 7
when Priority bit set to 1
Interrupt 0
when Priority bit set to 1
Low priority
process

Figure 7.1 Interrupt priority

## 7.1 Interrupt vector table

The interrupt controller contains a table of pointers to interrupt handlers. Each interrupt handler is represented by its work space pointer (**HandlerWptr**). The table contains a work space pointer for each level of interrupt.

The **HandlerWptr** gives access to the code, data and interrupt save space of the interrupt handler. The position of the **HandlerWptr** in the interrupt table implies the priority of the interrupt.

Run-time library support is provided for setting and programming the vector table.

## 7.2 Interrupt handlers

At any interruptible point in its execution the CPU can receive an interrupt request from the interrupt controller. The CPU immediately acknowledges the request.

In response to receiving an interrupt the CPU performs a procedure call to the process in the vector table. The state of the interrupted process is stored in the work space of the interrupt handler as shown in Figure 7.2. Each interrupt level has its own work space.

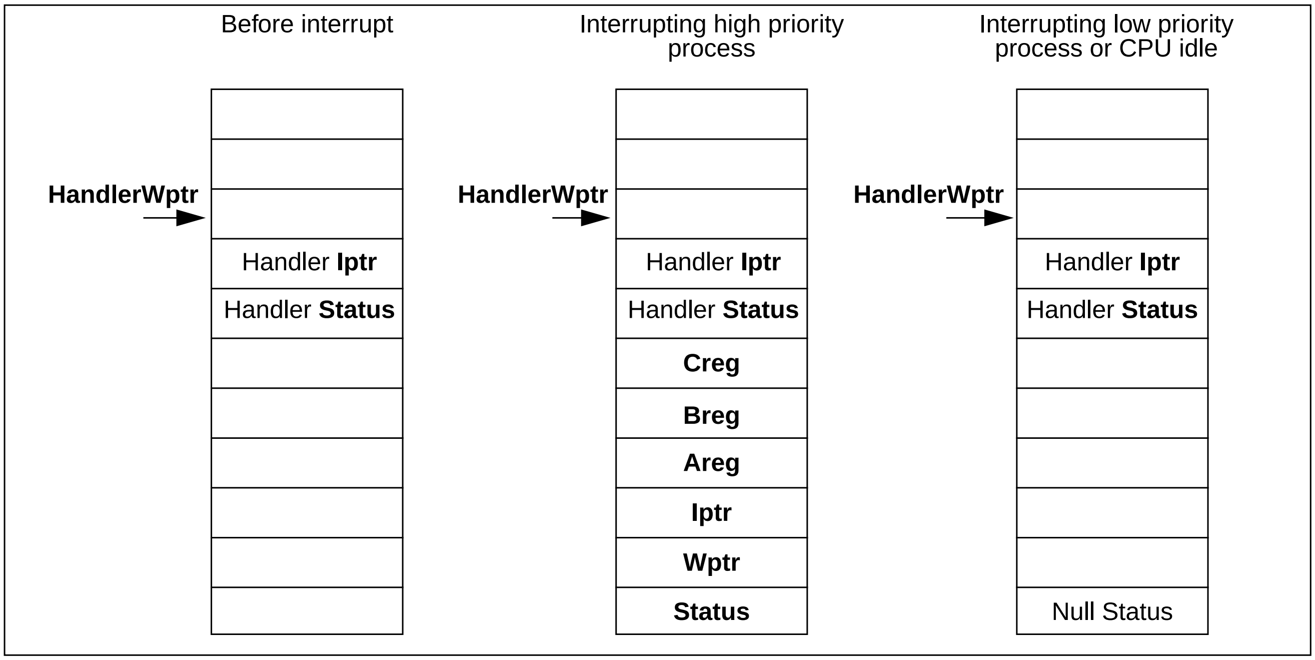


Figure 7.2 State of interrupted process

The interrupt routine is initialized with space below **HandlerWptr**. The **Iptr** and **Status** word for the routine are stored there permanently. This should be programmed before the **HandlerWptr** is written into the vector table.

The behavior of the interrupt differs depending on the priority of the CPU when the interrupt occurs. If an interrupt occurs when the CPU is running at high priority, and the interrupt is set at a higher priority than the high priority process queue, the CPU saves the current process state (**Areg**, **Breg**, **Creg**, **Wptr**, **Iptr** and **Status**) into the workspace of the interrupt handler. The value **HandlerWptr**, which is stored in the interrupt controller, points to the top of this work space. The values of **Iptr** and **Status** to be used by the interrupt handler are loaded from this work space and starts executing the handler. The value of **Wptr** is then set to the bottom of this save area.

If an interrupt occurs when the CPU is running at high priority, and the interrupt is set at a lower priority than the high priority process queue, no action is taken and the interrupt waits in a queue until the high priority process queue is empty (see section 7.4).

Interrupts always take priority over low priority processes. If an interrupt occurs when the CPU was idle or running at low priority, the **Status** is saved. This indicates that no valid process is running (*Null Status*). The interrupted processes (low priority process) state is stored in shadow registers. This state can be accessed via the *ldshadow* (load shadow registers) and *stshadow* (store shadow registers) instructions. The interrupt handler is then run at high priority.

When the interrupt routine has completed it must adjust **Wptr** to the value at the start of the handler code and then execute the *iret* (interrupt return) instruction. This restores the interrupted state from the interrupt handler structure and signals to the interrupt controller that the interrupt has completed. The processor will then continue from where it was before being interrupted.

## 7.3 Interrupt latency

The interrupt latency is dependent on the data being accessed and the position of the interrupt handler and the interrupted process. This allows systems to be designed with the best trade-off use of fast internal memory and interrupt latency.

## 7.4 Pre-emption and interrupt priority

Each interrupt channel has an implied priority fixed by its place in the interrupt vector table. All interrupts will cause scheduled processes of any priority to be suspended and the interrupt handler started. Once an interrupt has been sent from the controller to the CPU the controller keeps a record of the current executing interrupt priority. This is only cleared when the interrupt handler executes a return from interrupt (*iret*) instruction. Interrupts of a lower priority arriving will be blocked by the interrupt controller until the interrupt priority has descended to such a level that the routine will execute. An interrupt of a higher priority than the currently executing handler will be passed to the CPU and cause the current handler to be suspended until the higher priority interrupt is serviced.

In this way interrupts can be nested and a higher priority interrupt will always pre-empt a lower priority one. Deep nesting and placing frequent interrupts at high priority can result in a system where low priority interrupts are never serviced or the controller and CPU time are consumed in nesting interrupt priorities and not executing the interrupt handlers.

## 7.5 Restrictions on interrupt handlers

There are various restrictions that must be placed on interrupt handlers to ensure that they interact correctly with the rest of the process model implemented in the CPU.

1 Interrupt handlers must not deschedule.

5 Interrupt handlers must not execute communication instructions. However they may communicate with other processes through shared variables using the semaphore *signal* to synchronize.

6 Interrupt handlers must not perform 2D block move instructions.

7 Interrupt handlers must not cause program traps. However they may be trapped by a

scheduler trap.

## 7.6 Interrupt level controller

The interrupt level controller multiplexes twenty three incoming interrupt signals onto the eight interrupt inputs of the interrupt controller. In this way, it gives programmable control of the priority of the interrupts and extends the number of possible interrupts to twenty three.

There are twenty three interrupt signals to be handled by the interrupt subsystem. They may be generated by other on-chip subsystems or be received from external pins. Software assigns signal *n* to one of the 8 inputs to the interrupt controller by writing the priority of the required input in the register **Int***n***Priority**.

Thus each input of the interrupt controller responds to zero or more of the twenty three system interrupts. The interrupt level controller asserts interrupt output *p* when one or more of the input interrupts with programmed priority equal to *p* are high. It is level sensitive.

Where two or more system interrupts are assigned to one interrupt handler, the routine is able to ascertain the source of an interrupt by doing a device read from the **InputInterrupts** register and examining the bits that correspond to the system interrupts assigned to that handler.

## 7.7 Interrupt assignments

The interrupts from the internal peripherals and external pins on the STi5500 are assigned as shown in Table 7.1.

| Interrupt | Peripheral | Pin | Notes |
|---|---|---|---|
| 0 | Port 0 | | Compare function on PIO port. |
| 1 | Port 1 | | Compare function on PIO port. |
| 2 | Port 2 | | Compare function on PIO port. |
| 3 | Port 3 | | Compare function on PIO port. |
| 4 | Port 4 | | Compare function on PIO port. |
| 5 | SSC0 | | OR of signals SSC0TIR, SSC0RIR, SSC0EIR. |
| 6 | ASC3 | | OR of signals ASC3TIR, ASC3TBIR, ASC3RIR, ASC3EIR. |
| 7 | ASC2 | | OR of signals ASC2TIR, ASC2TBIR, ASC2RIR, ASC2EIR. |
| 8 | ASC1 | | OR of signals ASC1TIR, ASC1TBIR, ASC1RIR, ASC1EIR. |
| 9 | ASC0 | | OR of signals ASC0TIR, ASC0TBIR, ASC0RIR, ASC0EIR. |
| 10 | PWM and Capture | | OR of signals PWMInt, Capture[2:0]Int, Compare[2:0]Int. |
| 11 | Teletext | | Teletext DMA complete. |
| 12 | Transport Stream Demultiplexor | | |
| 13 | Reserved | | |
| 14 | Video decoder | | |
| 15 | Audio decoder | | |

Table 7.1 Interrupt assignments

| Interrupt | Peripheral | Pin | Notes |
|---|---|---|---|
| 16 | Reserved | | |
| 17 | Reserved | | |
| 18 | | **Irq0** | |
| 19 | | **Irq1** | |
| 20 | | **Irq2** | |

Table 7.1 Interrupt assignments

# 8 Memory map

The STi5500 has a 32-bit signed (twos complement) address space where the address ranges from **MinInt** (#80000000) at the bottom to **MaxInt** (#7FFFFFFF) at the top. 32-bit (four-byte) words are addressed by 30-bit word addresses, and a 2-bit byte-selector identifies the bytes in the word.

Memory is divided into areas with different memory characteristics and intended purposes. Some areas are dedicated to a specific purpose either because they contain memory-mapped devices or because they are reserved by the system.

Figure 8.1 shows the broad memory map arrangement, and Table 8.1 shows the details.

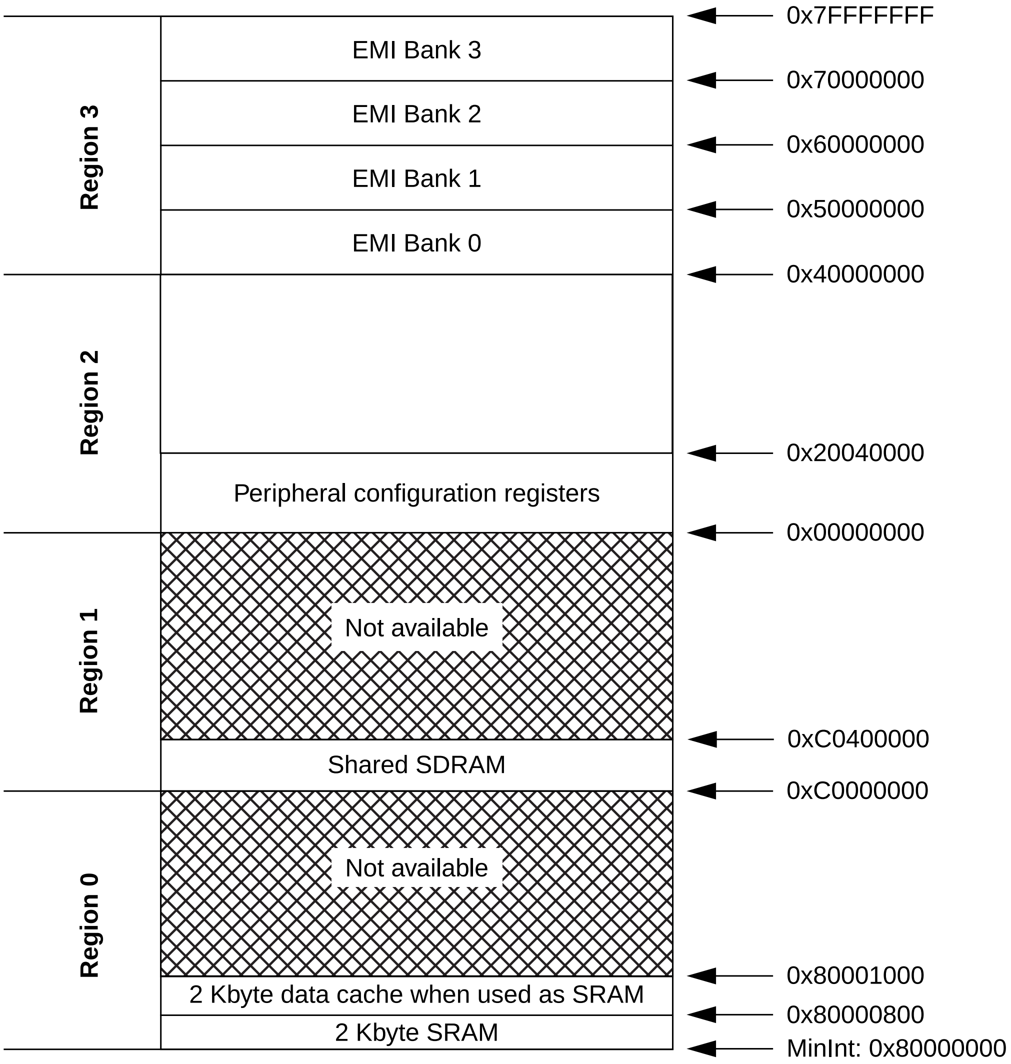


Figure 8.1 Memory map

The space is divided up for different uses as follows:

- The bottom 2 Kbytes, or optionally if the Data Cache is not used 4 Kbytes, is occupied by on-chip SRAM
- The 4 Mbyte area from 0xC0000000 to 0xC03FFFFF (in region 1) is for SDRAM, which is shared with the MPEG decoders.
- The area from 0x00000000 to 0x3FFFFFFF (region 2) is dedicated to memory-mapped or command-mapped on-chip peripherals. Memory-mapped on-chip or off-chip peripherals must be accessed via the Device Access Instructions rather than normal addressing.
- 0x40000000 to 0x7FFFFFFF (region 3) is for external memory and peripherals, accessed through the External Memory Interface (EMI).

## 8.1 System memory use

Addresses below **MemStart** are dedicated to processor use and should not be accessed directly but via the appropriate instructions. The address of **MemStart** can be obtained using the *ldmemstartval* instruction.

When booting from ROM, the system boots from the predefined location **BootEntry** (#7FFFFFFE) near the top of memory.

### 8.1.1 Subsystem channels memory

Each channel-based DMA subsystem is allocated a word of storage below **MemStart**. This is used by the processor to store information about the state of the channel. This information should not normally be examined directly, although debugging kernels may need to do so.

Interrupting DMA subsystems do not have a channel word allocated and rely on interrupts to perform synchronisation with the processes running on the processor.

### 8.1.2 Boot channel

The subsystem channel which is a link input channel is identified as a ‘boot channel’. When the processor is reset, and is set to boot from link, it waits for boot commands on this channel. In the case of STi5500 this is the OS-link channel **Link0**.

### 8.1.3 Trap handlers memory

The area of memory reserved for trap handlers is broken down hierarchically. Full details on trap handlers is given in the section on the CPU.

- Each high/low process priority has a set of trap handlers.
- Each set of trap handlers has a handler for each of four trap groups.
- Each trap group handler has a trap handler structure and a trapped process structure.
- Each of the structures contains four words.

The contents of these addresses can be accessed via *ldtraph*, *sttraph*, *ldtrapped* and *sttrapped* instructions.

### 8.1.4 Boot ROM

When the processor boots from ROM, it jumps to a boot program held in ROM with an entry point 2 bytes from the top of memory at #7FFFFFFE. These 2 bytes are used to encode a negative jump of up to 256 bytes down in the ROM program. For large ROM programs it may then be necessary to encode a longer negative jump to reach the start of the routine.

## 8.2 External Memory Space

The EMI decodes the top quarter of the address space into four banks into which different external memories and peripherals can be mapped.

When the STi5500 is in slave mode the EMI inverts address bits 21-19 for any access to Bank 3 to avoid a clash with the ROM address space used by the PowerPC.

Further details of the EMI can be found in Chapter 10.

## 8.3 Internal peripheral space

On-chip peripherals are mapped to addresses in the address range 0x00000000 to 0x3FFFFFFF. They can only be accessed by the device access instructions listed in Table 6.18.

Each on-chip peripheral occupies a 4 Kbyte block, as shown in Table 8.1. Table 8.2 lists the variables used elsewhere in this document to signify the bases of blocks of registers.

| Designation | Address (byte) | | Use |
|---|---|---|---|
| | Start | Finish | |
| MaxInt | #7FFFFFFF | | Boot jump offset byte 1 |
| BootEntry | #7FFFFFFE | | Boot entry point: jump offset byte 0 |
| | #70000000 | #7FFFFFFD | ST20 EMI bank 3:<br>Boot ROM |
| | #60000000 | #6FFFFFFF | ST20 EMI bank 2:<br>Uncommitted: User code/data/stack/peripherals |
| | #50000000 | #5FFFFFFF | ST20 EMI bank 1:<br>Uncommitted: User code/data/stack/peripherals |
| | #40000000 | #4FFFFFFF | ST20 EMI bank 0:<br>DRAM if present, else uncommitted:<br>User code/data/stack/peripherals |
| | #20027000 | #3FFFFFFF | RESERVED |
| | #20026000 | #20026FFF | Block move DMA controller peripheral* |
| | #20025000 | #20025FFF | RESERVED |
| | #20024000 | #20024FFF | Teletext DMA controller peripheral* |
| | #20023000 | #20023FFF | RESERVED |
| | #20022000 | #20022FFF | MPEGDMA2 (SDAV) controller peripheral* |
| | #20021000 | #20021FFF | MPEGDMA1 controller peripheral* |
| | #20020000 | #20020FFF | MPEGDMA0 controller peripheral* |
| | #20012000 | #2001FFFF | RESERVED |
| | #20011000 | #20011FFF | Interrupt level controller peripheral* |
| | #20010000 | #20010FFF | PIO4 controller peripheral* |
| | #2000F000 | #2000FFFF | PIO3 controller peripheral* |
| | #2000E000 | #2000EFFF | PIO2 controller peripheral* |
| | #2000D000 | #2000DFFF | PIO1 controller peripheral* |
| | #2000C000 | #2000CFFF | PIO0 controller peripheral* |
| | #2000B000 | #2000BFFF | PWM and counter controller peripheral* |
| | #2000A000 | #2000AFFF | RESERVED |
| | #20009000 | #20009FFF | SSC controller peripheral* |
| | #20008000 | #20008FFF | SmartCard1 clock generator peripheral* |
| | #20007000 | #20007FFF | SmartCard0 clock generator peripheral * |
| | #20006000 | #20006FFF | ASC3 controller peripheral* |
| | #20005000 | #20005FFF | ASC2 (SmartCard1) controller peripheral* |
| | #20004000 | #20004FFF | ASC1 controller peripheral* |
| | #20003000 | #20003FFF | ASC0 (SmartCard0) controller peripheral* |
| | #20002000 | #20002FFF | Transport stream demultiplexor registers* |
| | #20001000 | #20001FFF | RESERVED |
| | #20000000 | #20000FFF | Interrupt Controller* |

*Registers accessed via CPU device accesses

Table 8.1 STi5500 memory map

| Designation | Address (byte) | | Use |
|---|---|---|---|
| | Start | Finish | |
| | #00005000 | #1FFFFFFF | RESERVED |
| | #00004000 | #00004FFF | Cache configuration* |
| | #00003000 | #00003FFF | Diagnostic controller* |
| | #00002000 | #00002FFF | EMI configuration* |
| | #00001000 | #00001FFF | MPEG Data/Registers* |
| | #00000000 | #00000FFF | RESERVED |
| *Start of external memory* | #C0000000 | #FFFFFFFF | SDRAM: Video memory/user code/data/stack |
| | #80004000 | #BFFFFFFF | RESERVED |
| | #80000800 | #80000FFF | Internal SRAM if the data cache is not enabled. User code/data/stack |
| **MemStart** | #80000140 | #800007FF | Internal SRAM: <2Kbytes user code/data/stack |
| | #80000130 | #8000013F | Low priority Scheduler trapped process |
| | #80000120 | #8000012F | Low priority Scheduler trap handler |
| | #80000110 | #8000011F | Low priority SystemOperations trapped process |
| | #80000100 | #8000010F | Low priority SystemOperations trap handler |
| | #800000F0 | #800000FF | Low priority Error trapped process |
| | #800000E0 | #800000EF | Low priority Error trap handler |
| | #800000D0 | #800000DF | Low priority Breakpoint trapped process |
| | #800000C0 | #800000CF | Low priority Breakpoint trap handler |
| | #800000B0 | #800000BF | High priority Scheduler trapped process |
| | #800000A0 | #800000AF | High priority Scheduler trap handler |
| | #80000090 | #8000009F | High priority SystemOperations trapped process |
| | #80000080 | #8000008F | High priority SystemOperations trap handler |
| | #80000070 | #8000007F | High priority Error trapped process |
| | #80000060 | #8000006F | High priority Error trap handler |
| | #80000050 | #8000005F | High priority Breakpoint trapped process |
| **TrapBase** | #80000040 | #8000004F | High priority Breakpoint trap handler |
| | #80000038 | #8000003F | RESERVED |
| | #80000034 | #80000037 | Block move DMA controller channel out |
| | #8000002C | #80000033 | RESERVED |
| | #80000028 | #8000002B | MPEG2 (SDAV) DMA channel |
| | #80000024 | #8000002A | MPEG1 DMA channel |
| | #80000020 | #80000023 | MPEG0 DMA channel |
| | #80000014 | #8000001F | RESERVED |
| | #80000010 | #80000013 | Link0 (boot) input channel |
| | #80000004 | #8000000F | RESERVED |
| **MinInt** | #80000000 | #80000003 | Link0 output channel |

*Registers accessed via CPU device accesses

Table 8.1 STi5500 memory map

| Variable | Value | Block |
|---|---|---|
| *ASC0BaseAddress* | 0x20003000 | Asynchronous serial controller (ASC) 0. |
| *ASC1BaseAddress* | 0x20004000 | Asynchronous serial controller (ASC) 1. |
| *ASC2BaseAddress* | 0x20005000 | Asynchronous serial controller (ASC) 2. |
| *ASC3BaseAddress* | 0x20006000 | Asynchronous serial controller (ASC) 3. |
| *AudioBaseAddress* | 0x00001200 | MPEG audio decoder. |
| *BMBaseAddress* | 0x20026000 | Block move DMA controller. |
| *CacheBaseAddress* | 0x00004000 | Cache configuration. |
| *DCUBaseAddress* | 0x00003000 | Diagnostic controller unit (DCU). |
| *DENCBaseAddress* | 0x00001600 | PAL/NTSC digital encoder. |
| *EMIBaseAddress* | 0x00002000 | External memory interface (EMI). |
| *IntControllerBase* | 0x20000000 | Interrupt controller. |
| *InterruptLevelBase* | 0x20011000 | Interrupt level controller. |
| *MPEGDMA0BaseAddress* | 0x20020000 | MPEG DMA0 controller. |
| *MPEGDMA1BaseAddress* | 0x20021000 | MPEG DMA1 controller. |
| *MPEGDMA2BaseAddress* | 0x20022000 | MPEG DMA2 (SDAV) controller. |
| *PIO0BaseAddress* | 0x2000C000 | PIO port 0 controller. |
| *PIO1BaseAddress* | 0x2000D000 | PIO port 1 controller. |
| *PIO2BaseAddress* | 0x2000E000 | PIO port 2 controller. |
| *PIO3BaseAddress* | 0x2000F000 | PIO port 3 controller. |
| *PIO4BaseAddress* | 0x20010000 | PIO port 4 controller. |
| *PWMBaseAddress* | 0x2000B000 | PWM and counter module. |
| *SmartCard0BaseAddress* | 0x20007000 | SmartCard interface 0. |
| *SmartCard1BaseAddress* | 0x20008000 | SmartCard interface 1. |
| *SSCBaseAddress* | 0x20009000 | Synchronous serial controller (SSC) 0. |
| *SubPictureBaseAddress* | 0x00001400 | Sub-picture decoder. |
| *TransportDemuxBase* | 0x20002000 | Transport stream demultiplexor. |
| *TtxtBaseAddress* | 0x20024000 | Teletext interface. |
| *VideoBaseAddress* | 0x00001000 | MPEG video decoder. |

Table 8.2 Register block base variables

# 9 Memory

Memory is normally accessed by the load, store, block move and channel instructions. These will use data cache if it is enabled, and do not guarantee the order of accesses to different addresses. The device access instructions listed in Table 6.18 should be used when there is a need to bypass the data cache in a cacheable area, or if there is a need to know when a write occurs to an external device or memory area.

## 9.1 External memory

### 9.1.1 EMI accessible memory

The EMI decodes region 3 of the address space into four banks, into which different external memories and peripherals can be mapped. Further details of the EMI can be found in Chapter 10. One of the banks supports DRAM and one bank is normally used for boot ROM.

- The locations 0x40000000 to 0x4FFFFFFF are generally used for DRAM, but may be used for any external memory or peripherals.
- The locations 0x50000000 to 0x6FFFFFFF may be used for any external memory or peripherals except DRAM.
- The locations 0x70000000 to 0x7FFFFFFF may be used for any external memory or peripherals except DRAM, but are generally used for boot ROM. When booting from ROM, the system boots from the predefined location **BootEntry** (0x7FFFFFFE) near the top of memory space.

Accessing some areas of memory causes special access characteristics (strobes etc.) to be generated depending on the way the EMI is programmed.

The EMI provides address decoding, address and data buses, timing strobes, enabling signals and refresh where appropriate.

### 9.1.2 SDRAM

SDRAM occupies the first 32 Mbits of region 1, and is shared with the MPEG decoders. OSD bitmaps, for example, are stored in this memory.

For details of the SDRAM interface configuration and set-up, refer to the register manual.

## 9.2 On-chip SRAM memory

This internal memory module, known as on-chip memory, contains

- 2 Kbytes of dedicated SRAM, mapped into the lowest 2 Kbytes of memory space from **MinInt** (0x80000000) extending upwards, as shown in Figure 8.1.
- 2 Kbytes which can be configured as SRAM or as data cache. When it is configured as SRAM, it is mapped to the 2 Kbytes immediately above the dedicated SRAM, i.e. from 0x80000800 upwards, as shown in Figure 8.1. The default configuration is SRAM.

Part of the lowest 2 Kbytes of memory is committed to system use; see section 8.1 and Table 8.1 for details. The remainder of the lowest 2 Kbytes of memory is uncommitted and can be used to store on-chip data, stack or code for time-critical routines.

Locations between 0x80001000 (or 0x80000800 if data cache is used) and 0xBFFFFFFF should not be addressed.

## 9.3 Caching

Cache can be used to reduce the average access delay imposed on the CPU when it accesses a memory location to read or write. Some locations should not be cached, for example those to which other modules have direct memory access.

The STi5500 cache subsystem provides:

- 2 Kbytes of direct-mapped write-back data cache;
- 2 Kbytes of direct-mapped read-only instruction cache.

The data cache may be configured as memory. If it is used as data cache, the region bounded by the addresses 0x80000800 to 0x80000FFF in main memory should not be used.

The instruction cache is identical in operation to the data cache, except that it is read-only and cannot be configured as SRAM.

The cache configuration is held in memory-mapped registers. The registers must be accessed using the device access instructions.

Device access instructions can also be used to force access to external memory without going through the cache. These instructions can be used to solve any cache coherency issues. Device writes do not change the value in the cache.

Registers are provided to configure areas of memory to be cacheable or non-cacheable for data access, as described in section 9.4.5.

Note that the correct cache initialization sequences, described in section 9.3.2, must be used before the caches are enabled.

### 9.3.1 Outline of Operation

The cache is four 32-bit words (16 bytes) wide and 128 lines (2 KBytes, 512 words) high. It is direct-mapped (sometimes called *one way set associative*). This is shown in Figure 9.1.

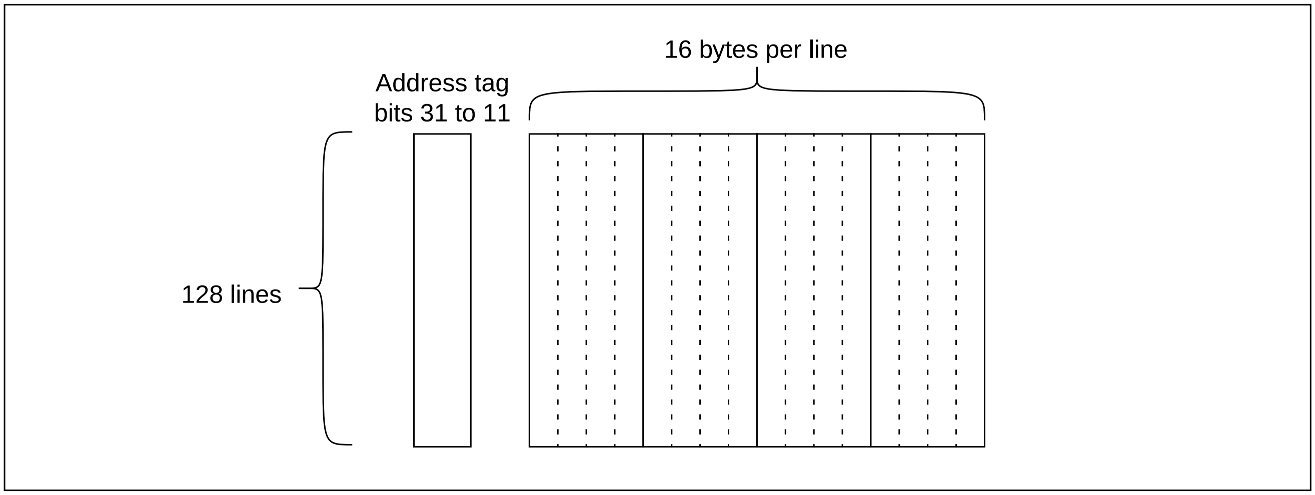


Figure 9.1 2 Kbyte data or instruction cache

Each line of the cache can only store data from specific four-word sections of memory at 2 Kbyte intervals, with the bottom line of the cache coinciding with the 4 words just above each 2 Kbyte boundary. Thus the line number of the cache pinpoints the four-word section of memory within a 2 Kbyte block, i.e. bits 4 to 10 of the address. The 21 most significant bits of the address selects the 2 Kbyte block. These 21 bits are stored in 128 tag registers, with one tag register corresponding to each cache lines. The significance of the parts of the address when using the cache are shown in Figure 9.2

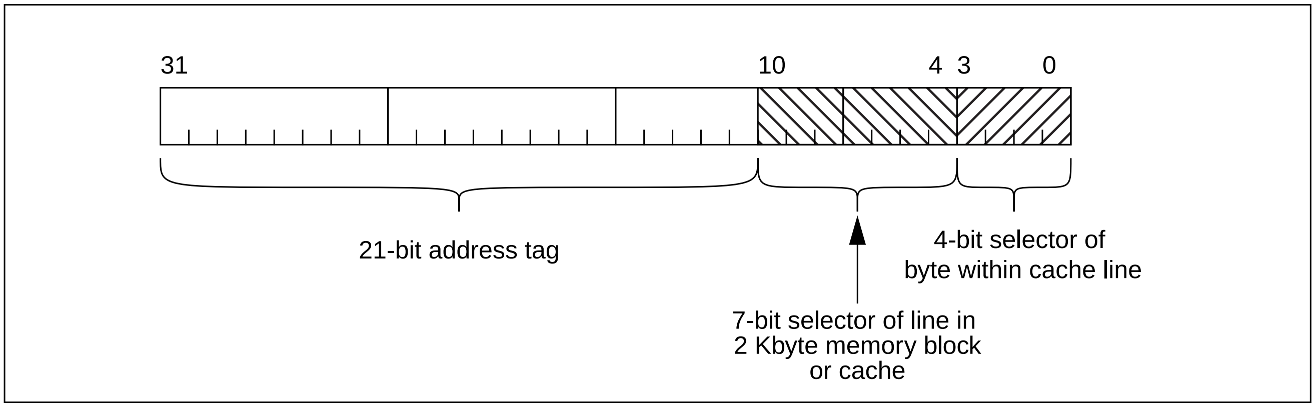


Figure 9.2 Address fields when using cache

If a request is made to access a cacheable memory location, and a copy of that location is held in cache, then the access is said to have made a cache hit. A hit is identified by comparing the address bits 11 to 31 with the address tag for the cache line given by the address bits 4 to 10. If the cache is hit, then the access is completed by the cache subsystem. If the cache is missed, the appropriate cache line is written back to memory, and if necessary the new location in memory is read into that cache line. All cache reads and writes to memory are complete lines because of the efficiency of accessing the memory in burst mode.

### 9.3.2 Cache initialization

Before the caches are enabled, they must be correctly initialized. To do this the cache must first be invalidated before it is accessed. To ensure this occurs, the invalidate bit of each cache must be set with the cache disabled and then the enable bit set to enable the cache.

This sequence has the effect of forcing a cache to be invalid, which initializes the cache state before any other accesses are considered by the cache.

## 9.4 Cache subsystem control registers

The cache subsystem can be controlled by registers which are mapped into the device address space. The registers are grouped in a 4 Kbyte block, with the base of the block at the address *CacheBaseAddress*. The value of *CacheBaseAddress* is given in the *Memory Map* chapter. The addresses of the registers are given in the tables as offsets from this address. The cache control registers are listed in Table 9.1.

| Register | Offset | Bits | Access | Function | Reset value |
|---|---|---|---|---|---|
| **CacheControl** | 0x000 | 8 | R/W | Cacheability of address range 0xC0000000 to 0xC007FFFF and 0xC0200000 to #C027FFFF. | 0 |
| **SelectCache** | 0x100 | 2 | W | Enable cache. | 0 |
| **InvalidateDCache** | 0x200 | 1 | W | Invalidate the data cache. | - |
| **InvalidateICache** | 0x300 | 1 | W | Invalidate the instruction cache. | - |
| **FlushDCache** | 0x400 | 1 | W | Flush the data cache. | - |
| **CacheControlLock** | 0x500 | 1 | R/W | Lock the **CacheControl**, **SelectCache** and **CacheControlLock** registers. | 0 |

Table 9.1 Cache control registers

The cache subsystem registers enable the caches, control cache functions such as flushing and invalidation, and are used to mark sections of memory space as cacheable or not cacheable. Registers should be accessed using the device access instructions.

The **CacheControlLock** must be 0 before cache can be enabled or memory can be made cacheable. After changing these registers, the **CacheControlLock** should be set to 1. Once this lock is set it cannot be cleared except by a reset. It is not recommended to change the cache configuration other than at reset.

### 9.4.1 Selecting cache

It is possible to select either data cache or an extra 2 Kbyte of on-chip SRAM. This is done by writing to the **DCacheNotSRAM** bit of the **SelectCache** register. The default is to select the extra on-chip SRAM. It is not recommended to change the selection other than during booting of the application. To select data cache mode, set the **DCacheNotSRAM** bit to 1. Do not access locations 0x80000800 to 0x80000FFF when using the data cache.

The instruction cache is enabled by setting **EnableICache** to 1 in the **SelectCache** register.

All cache should be invalidated before being enabled, as described in section 9.3.2. When cache is enabled, the cache contents will be random and must be invalidated by setting the invalidate bit first before enabling the cache.

Changing from SRAM to data cache should normally only be performed during the initialization stage of an application. However, if it is necessary to do so at other times, it is essential to invalidate the cache contents by setting the invalidate bit first and then enabling the cache.

It is not recommended to change selection from data cache to SRAM during operation. However, if it is necessary to do so, it is essential to flush the cache to maintain memory integrity before making the change.

| **SelectCache** | **CacheBaseAddress + 0x100** | **Write only** |
|---|---|---|
| **Bit** | **Bit field** | **Function** |
| 0 | **DCacheNotSRAM** | Select the configuration for data cache memory.<br>0 SRAM (default).<br>1 Data cache. |
| 1 | **EnableICache** | Enable the instruction cache.<br>0 Disabled.<br>1 Enabled. |

Table 9.2 **DCacheNotSRAM** register format

### 9.4.2 Invalidating cache

Invalidating a cache marks every line as not containing valid data.

#### Data cache

The data cache is invalidated by setting the **InvalidateDCache** register to 1. This register is automatically reset to 0 on completion of the task.

Any memory accesses that are cacheable which are started before the data cache invalidation is complete will be blocked until it is completed.

| **InvalidateDCache** | **CacheBaseAddress + 0x200** | **Write only** |
|---|---|---|
| **Bit** | **Bit field** | **Function** |
| 0 | **Invalidate** | Select the configuration for data cache memory.<br>0 No action.<br>1 Invalidate the data cache. |

Table 9.3 **InvalidateDCache** register format

#### Instruction cache

The instruction cache is invalidated by setting the **InvalidateICache** register to 1. This register is automatically reset to 0 on completion of the task.

Any instruction fetches that are cacheable and were started before completion of the invalidation of the instruction cache, will be blocked until it is completed.

| **InvalidateICache** | **CacheBaseAddress + 0x300** | **Write only** |
|---|---|---|
| **Bit** | **Bit field** | **Function** |
| 0 | **Invalidate** | Invalidate the instruction cache.<br>0 No action.<br>1 Invalidate the instruction cache. |

Table 9.4 **InvalidateICache** register format

### 9.4.3 Flushing the data cache

Flushing the cache means forcing a write-back to memory of every dirty line in the cache. A dirty line is a line of cache that has been written to since it was loaded or last written back. Only the data cache can be flushed; the instruction cache never needs flushing since it is read only.

To flush the data cache, set the **FlushDCache** register to 1. It is automatically reset to 0 on completion of the task. Any memory accesses that are cacheable which were started before the flush of the data cache is complete, will be blocked until it is completed.

| **FlushDCache** | **CacheBaseAddress + 0x400** | **Write only** |
|---|---|---|
| **Bit** | **Bit field** | **Function** |
| 0 | **Flush** | Flush the data cache.<br>0 No action.<br>1 Flush the data cache. |

Table 9.5 **FlushDCache** register format

### 9.4.4 Lock register

The cache configuration can be locked by writing a 1 to the **CacheControlLock** register bit. Reset of this flag is only performed by a hardware reset. This bit should be set to 1 after all the cache configuration registers have been written.

| **CacheControlLock** | **CacheBaseAddress + 0x500** | **Read/write** |
|---|---|---|
| **Bit** | **Bit field** | **Function** |
| 0 | **Lock** | Lock the **CacheControl**, **SelectCache** and **CacheControlLock** registers.<br>0 Unlocked, i.e. registers can be written.<br>1 Locked, i.e. registers can only be read. |

Table 9.6 **CacheControlLock** register format

### 9.4.5 Cacheable and non-cacheable memory locations

It may be desirable for some locations in memory to be not cached. For example, where other units have direct memory access, the cache could get out of step with the memory, i.e. the cache could become incoherent.

Some areas of memory are predefined to be cacheable, some are predefined to be not cacheable, and other areas may be programmed to be cacheable by the data cache using the configuration registers. The cacheability of an area of memory by the data cache may be different from the cacheability by the instruction cache. Cacheability by the instruction cache is predefined.

Table 9.7 summarizes the cacheability of different areas of memory. Table 9.9, illustrated by Figure 9.3 shows the programmable cacheability by the data cache.

Setting one bit in the **CacheControl** register makes two 64 Kbyte blocks of SDRAM cacheable, one block near the bottom of each half of SDRAM space.

| Region | Memory Range | Data cache | Instruction cache |
|---|---|---|---|
| Region 0 | 0x80000000 - 0xBFFFFFFF | Only the bottom 2 Kbytes of SRAM. | Not cacheable. |
| Region 1 | 0xC0000000 - 0xFFFFFFFF | Not cacheable except where defined by the **CacheControl** register. | Fully cacheable. |
| Region 2 | 0x00000000 - 0x3FFFFFFF | Not cacheable. | Fully cacheable. |
| Region 3 | 0x40000000 - 0xFFFFFFF | Fully cacheable. | Fully cacheable. |

Table 9.7 Memory cacheability

| **CacheControl** | **CacheBaseAddress + 0x000** | **Read/write** |
|---|---|---|
| **Bit** | **Bit field** | **Function** |
| 0-7 | **Cacheable0-7** | Set the cacheability by the data cache of the two address ranges given in Table 9.9.<br>0 Not cacheable.<br>1 Cacheable. |

Table 9.8 **CacheControl** register format

| Bit | Lower half of SDRAM | | | Upper half of SDRAM | | |
|---|---|---|---|---|---|---|
| | Block start | Block end | Block size | Block start | Block end | Block size |
| 0 | 0xC0000000 | 0xC000FFFF | 64 Kbytes | 0xC0200000 | 0xC020FFFF | 64 Kbytes |
| 1 | 0xC0010000 | 0xC001FFFF | 64 Kbytes | 0xC0210000 | 0xC021FFFF | 64 Kbytes |
| 2 | 0xC0020000 | 0xC002FFFF | 64 Kbytes | 0xC0220000 | 0xC022FFFF | 64 Kbytes |
| 3 | 0xC0030000 | 0xC003FFFF | 64 Kbytes | 0xC0230000 | 0xC023FFFF | 64 Kbytes |
| 4 | 0xC0040000 | 0xC004FFFF | 64 Kbytes | 0xC0240000 | 0xC024FFFF | 64 Kbytes |
| 5 | 0xC0050000 | 0xC005FFFF | 64 Kbytes | 0xC0250000 | 0xC025FFFF | 64 Kbytes |
| 6 | 0xC0060000 | 0xC006FFFF | 64 Kbytes | 0xC0260000 | 0xC026FFFF | 64 Kbytes |
| 7 | 0xC0070000 | 0xC007FFFF | 64 Kbytes | 0xC0270000 | 0xC027FFFF | 64 Kbytes |

Table 9.9 Blocks of memory which may be programmed to be data cacheable

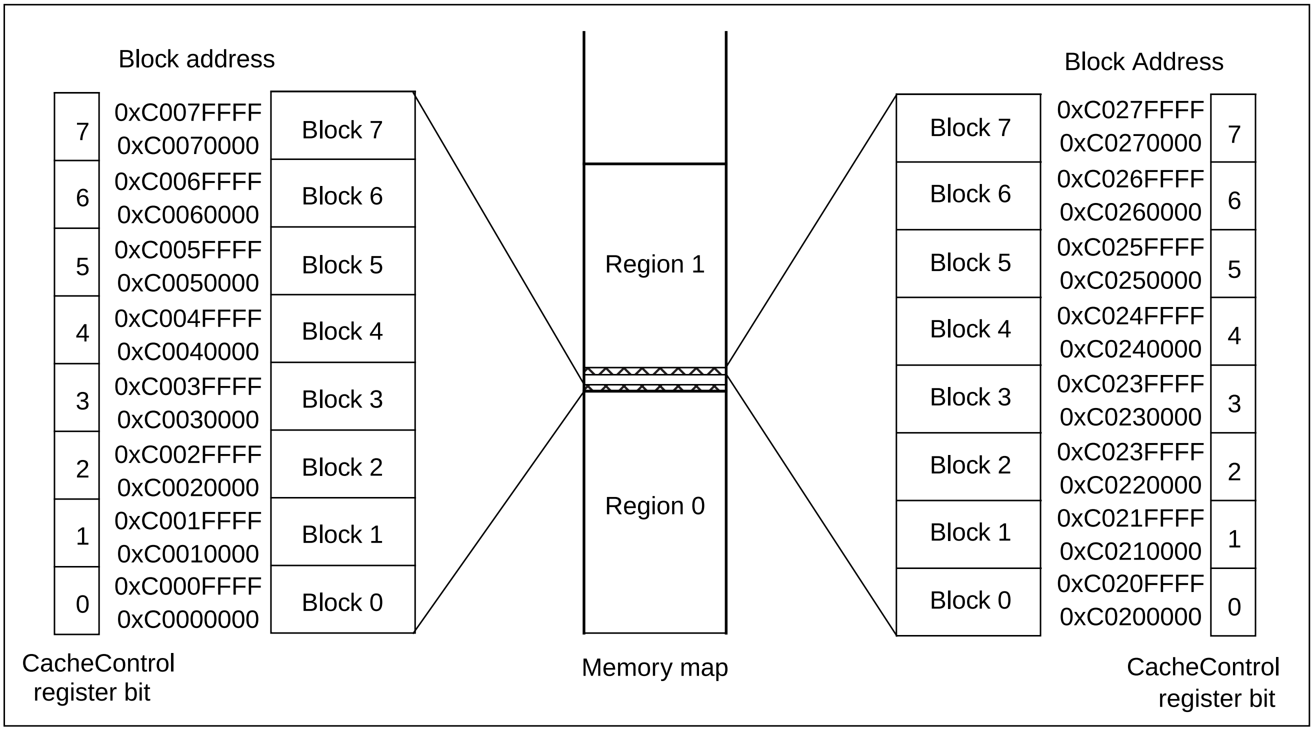


Figure 9.3 Cacheable memory areas in Region 1

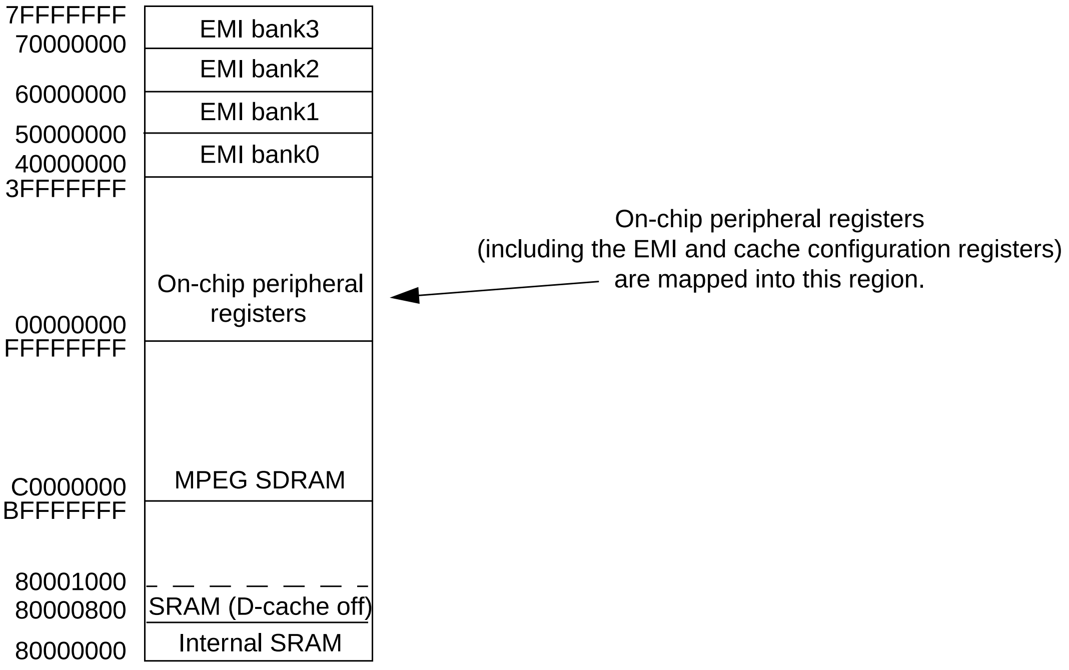
# 10 External memory interface

The External Memory Interface (EMI) controls the movement of data between the STi5500 and off-chip memory. It is designed to support memory subsystems with minimal (often zero) external support logic.

The EMI can access a 16 Mbyte physical address space (greater if DRAM is used) in four general purpose memory banks. The EMI supports the memory subsystems required in most set top receiver applications, including 16-bit DRAM devices, with zero external support logic.

The interface can be configured for a wide variety of timing and decode functions through configuration registers.

The EMI maps external memory into the top quarter of the address space and is partitioned into four banks with each bank occupying one sixteenth of the total address space (see Figure 10.1). This allows the implementation of mixed memory systems with support for DRAM, SRAM, EPROM, and I/O. The timing of each of the four memory banks can be selected separately, with different device types being placed in each bank with no external hardware support.



Addresses shown are physical addresses.

Figure 10.1 Memory allocation

The EMI supports two distinct types of memory, called device types:

- DRAM type with a multiplexed row and column address which is used to support fast page mode DRAM or other devices with multiplexed rows and columns;
- SRAM or peripheral type which is used to support SRAMs, peripherals, EPROM or Flash ROMs.

EMI bank 0 can support either type, while banks 1, 2 and 3 only support SRAM or peripheral type. Only 16-bit wide devices are supported.

As the banks are of a fixed size, range checking of addresses is not possible. This means that software tools must be aware of the physical external memory capacity. The behavior of some of the strobes depends on whether the bank being accessed has been configured as DRAM or SRAM / peripheral.

In this chapter a *cycle* is one processor clock cycle and a *phase* is one half of the duration of one processor clock cycle.

## 10.1 Pin functions

This section describes the functions of the external memory interface pins. A signal name prefixed by **not** indicates active low.

### Data0-15

The data bus transfers 16-bit data items. The least significant bit of the data bus is always **Data0**, and the most significant bit is **Data15**.

### Adr1-21

The address bus may be operated in both multiplexed and non-multiplexed modes. When bank 0 is configured to device type DRAM then the internally generated 32-bit address is multiplexed as row and column addresses through the external address bus.

### not_WE0-1

The EMI uses 2-byte word addressing and two byte enable strobes are provided, one for each byte.16-bit wide memory is defined as an array of 2-byte words, with 22 address bits selecting a 2-byte word and **not_WE0-1** selecting a byte within the word. **not_WE0** enables the least significant byte, named byte 0, containing data bits 0 to 7. This pin should be connected to the enable for **Data0-7**. Similarly, **not_WE1** enables the most significant byte, named byte 1, containing data bits 8 to 15. This pin should be connected to the enable for **Data8-15**. Other bus masters must not drive the same data pins during a write.

For banks configured for SRAM or peripherals, the **not_WE** strobes are fully programmable. They may be used as data enable strobes with the same timing and may be configured to be active on read cycles, write cycles, or both read and write cycles. For banks configured as DRAM, the **not_WE** strobes are directly related to the **not_Cas** strobe timing for that bank and are active during write cycles only.

### not_Ras0-1

EMI banks 0 is capable of supporting DRAM devices. Furthermore this bank may be sub-decoded into two sub-banks. The stobes **not_Ras0-1** are used as the DRAM RAS strobes to this bank or these sub-banks.

If bank 0 is not configured for device type DRAM, then the **not_Ras0** strobe is used as a chip select for this banks.

Table 10.1 summarizes the behavior of the **not_Ras0-1** strobes for bank 0.

| Bank configuration | not_Ras0 pin | not_Ras1 pin |
|---|---|---|
| Bank 0 DRAM with no sub-decoding | Bank 0 RAS strobe | Unused |
| Bank 0 DRAM with two sub-banks | Bank 0 sub-bank 0 RAS strobe | Bank 0 sub-bank 1 RAS strobe |
| Bank 0 contains SRAM / peripheral | Bank 0 chip select strobe | Unused |

Table 10.1 RAS pin functionality for bank 0

### not_Cas0-1

The two CAS strobes **not_Cas0-1** are only used for bank 0 when configured for DRAM-type devices. The CAS strobes can be programmed to be in one of two modes.

- Bank mode in which only one CAS strobe is used for the entire bank and sub-banks (if any).
- Byte mode in which each CAS strobe is used as a byte-decoded CAS strobe and can be used across both banks (and any sub-banks).

Byte mode is used to support 16-bit wide DRAMs or DRAM modules that provide multiple CAS strobes, one for each byte, and a single write signal to allow byte write operations.

The alternative type of DRAMs that has multiple write signals, one for each byte, and a single CAS to allow byte write operations or banks that are constructed from 1, 4, or 8-bit wide DRAMs can be interfaced using bank mode.

*CAS strobes in bank mode*

If bank 0 is set to DRAM device type with bank mode selected, then **not_Cas0** is the sole CAS strobe for bank 0. Unused CAS strobes remain inactive during an access.

*CAS strobes in byte mode*

For banks containing DRAM that requires byte decoded CAS strobes, one programmable CAS strobe is allocated to each byte. Each of the CAS strobes in this mode will have the timing programmed into the CAS timing configuration registers of bank 0, if they are active during that cycle. Byte mode CAS strobes are active during an access if the byte corresponding to the strobe is being accessed.

During refresh cycles, all CAS strobes will go low at the start of the cycle and remain low until the end of the cycle.

In byte mode, **not_Cas0** enables **Data0-7** and **not_Cas1** enables **Data8-15**. Only the CAS strobes that enable bytes which are being accessed will be active during an access cycle.

*Mixing bank and byte mode*

For full flexibility bank 0 features CAS mode (byte or bank mode) support. Table 10.2 gives a full listing of the active strobes for each mode.

| Bank configuration | not_Cas0 | not_Cas1 |
|---|---|---|
| Bank mode | Active | Unused |
| Byte mode | Active **Data0-7** | Active **Data8-15** |

Table 10.2 Active strobes in bank and byte mode

### not_CE0-3

The **not_CE0-3** strobes act as the chip select strobes for banks 0 to 3 respectively of the EMI. Bank 3 usually, though not necessarily, contains the system ROM.

### MemWait

Wait states can be generated by taking **MemWait** high. **MemWait** is sampled during SRAM or peripheral accesses only.

**MemWait** retains the state of any strobe during the cycle after the one in which it was asserted until it is deasserted. When **MemWait** is de-asserted the access continues as programmed by the configuration interface. The **MemWait** signal must be synchronous with the **ProcClockOut** clock.

### not_OE

The behavior of the **not_OE** signal depends on the type of memory being accessed. If the access is to a bank configured for DRAM then the **not_OE** strobe is active only during a read access when it is asserted low CASe1Time after the start of **CASTime**, and deasserted high at the end of **CASTime**. For accesses to configured as SRAM / peripheral the **not_OE** strobe is programmable and will behave according to the values in the **EMIConfigData** registers for that bank.

### ReadnotWrite

This signal indicates whether the current cycle is a read or a write cycle. During writes, the signal is asserted low at the beginning of the access (i.e. at the start of **RASTime** for DRAM banks and at the start of **CSTime** for SRAM / peripheral banks) and deasserted high at the end of the access (end of **CASTime** / **CSTime**). At all other times this signal is held high.

### ProcClockOut

This is a reference signal for external bus cycles, which oscillates at the processor clock frequency.

## 10.1.1 External bus cycles

The external memory interface is designed to provide efficient support for dynamic memory and other devices such as static memory and IO devices. This flexibility is provided by allowing the required wave-forms to be programmed via configuration registers (see section 10.2).

Memory is byte addressed, with 16-bit words aligned on 2-byte boundaries.

During read cycles byte level addressing is performed internally by the STi5500. The EMI can read bytes or words. The width of the bank is fixed at 16 bits.

During write cycles the STi5500 uses the **not_WE0-1** strobes to perform addressing of bytes. If a particular byte is not to be written then the corresponding data outputs are tristated. Writes can be less than the size of the bank.

The internally generated address is indicated on pins **Adr1-21**. The least significant bit of the data bus is always **Data0**.

The following sections describe the access cycles for the two device types supported, DRAM and SRAM or peripherals

### 10.1.2 DRAM access cycles

DRAM access cycles are supported in Bank 0 only when it is set to device type DRAM.

A DRAM memory access cycle consists of a number of defined periods or times, as shown in Figure 10.2. All of the named times shown in this diagram together with other parameters such as RAS address shift and page size are programmable to suit a wide variety of DRAM types.

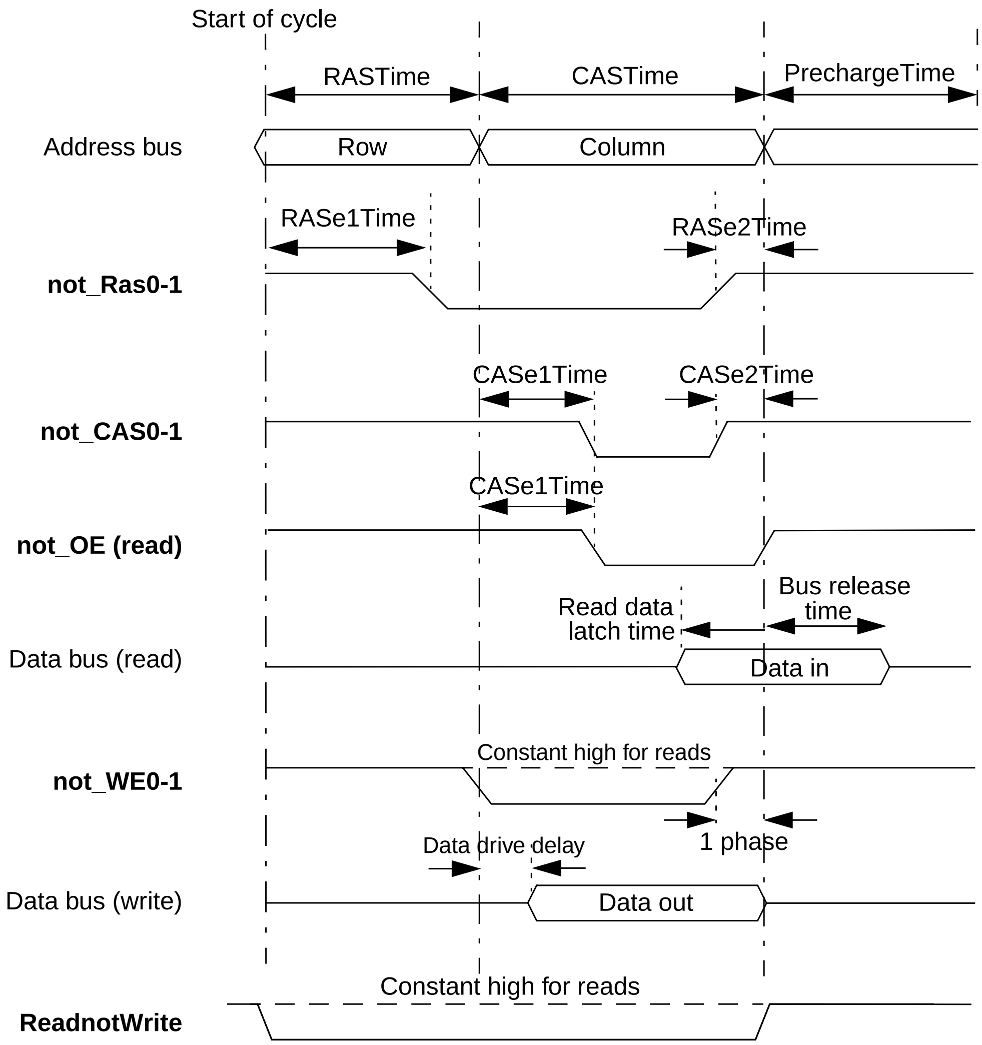


Figure 10.2 DRAM memory cycle

**RASTime** and **CASTime** are consecutive. The **CASTime** can be followed by concurrent Precharge and BusRelease times.

Thus for DRAM, these times are used for RAS address latching, CAS address latching, RAS precharge and output driver tristate times respectively. For consecutive access to the same bank of DRAM, **RASTime** will only occur when there is a page miss. The next access will not commence until the **PrechargeTime** for a previous access to the same bank has completed. During the **RASTime**, a transition can only be programmed on the RAS strobes.

During the **CASTime** the CAS strobes and either the byte-enable or **not_OE** strobes are active. The address is output on the address bus without being RAS shifted. Write data is valid during **CASTime**. Read data is latched into the interface at the point defined by the **LatchPoint** bit in the **EMIConfigData3** register for the bank being accessed.

The **PrechargeTime** and **BusReleaseTime** commence concurrently at the end of the **CASTime**. A **PrechargeTime** will occur, and the active **not_Ras** strobe will be taken high if:

- the next access is to the same bank but to a different row address.
- the next access is to a different bank.

Having two sub banks doubles the page size of the bank.

The **BusReleaseTime** runs concurrently with the **PrechargeTime** and will occur if:

- the current cycle is a read and the next cycle is a write.
- the current cycle is a read and the next cycle is a read from a different bank.

The **BusReleaseTime** is provided to allow an accessed device to float to a high impedance state.

### Page mode

DRAM pages are delineated using the **RASBits** configuration parameter. These bits are used as an address mask for comparison with the previous DRAM address. If an access is requested by an internal subsystem of the STi5500 to a DRAM bank while a DRAM access is in progress, the new address is compared to the current access address. If the row addresses are the same, the access may proceed as a page mode access. There is no specific configuration bit to select pagemode DRAM. If all the **RASBits** are set to 0, then no page hits will be caused and normal DRAM RAS/CAS cycles will always be produced.

A page mode access does not include the **RASTime** period. The **not_Ras** strobe is not taken high before commencing the page mode access. If the current access is a read and the page mode access is due to be a write, a **BusReleaseTime** is inserted as shown in Figure 10.3. The **not_Ras** strobe is held low during this period.

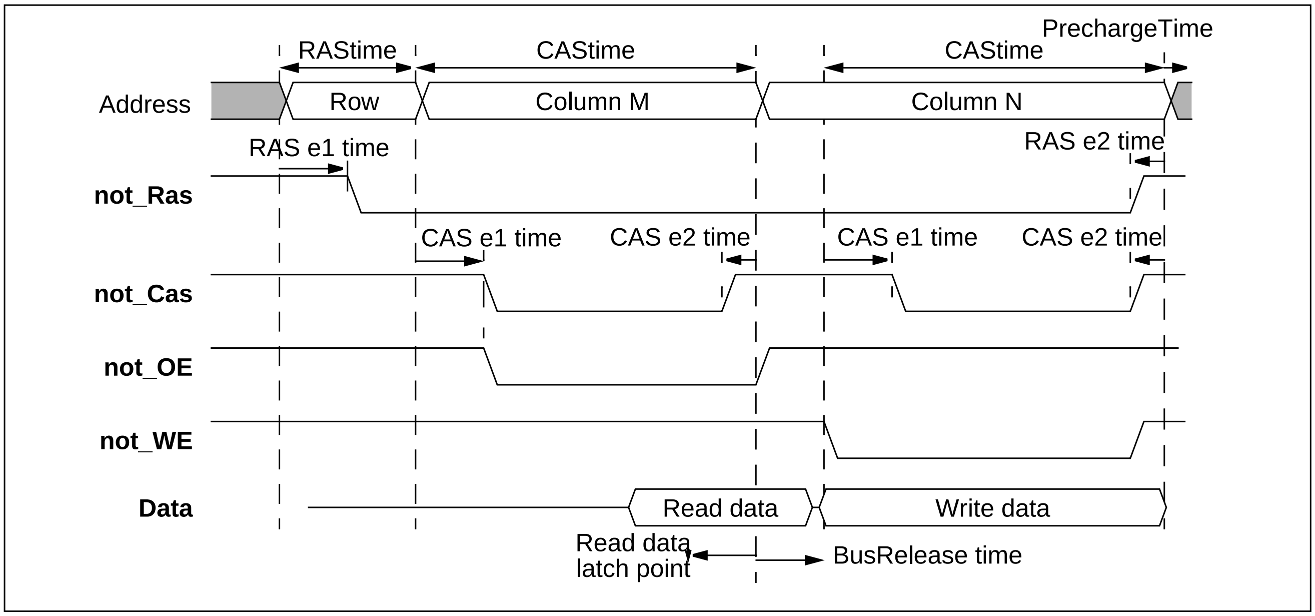


Figure 10.3 Read followed by page mode write

If the bank has been sub-decoded, the sub-bank selection address bits must be included in the comparison, so the **RASBits** corresponding to these addresses must be set.

For example, if the DRAM bank is composed of two 256k × 16 devices, the sub-bank selection address bit is $A_{19}$, so the **RASBits** corresponding to address bits $A_{19}$-$A_{10}$ must be set.

When page mode is active, the **RASe2time** must be programmed to zero. Future upgrades may relax this constraint; it is not considered essential now.

| Sub-banks | Sub-bank size | Sub-bank selection pin | RAS strobe selection |
|---|---|---|---|
| 2 | 256K<br>1M<br>4M<br>16M | **Adr19**<br>**Adr21**<br>**Adr23**<br>**Adr25** | 0 = **not_Ras0**<br>1 = **not_Ras1** |

Table 10.3 Address decoding

## Refresh

DRAM banks are periodically refreshed at intervals specified by the **RefreshInterval** configuration parameter.

The **not_Cas** strobes are taken low at the beginning of the refresh time. The position of the RAS falling edge (**RASedge**) is programmable and the minimum width of the CAS pulse is the sum of the **RASTime** and **CASTime** values specified for random access. If there is more than one bank of DRAM the refresh configuration will then be taken from the lowest numbered bank configured as DRAM.

All sub-banks are refreshed in the same access and a cycle is inserted between each sub-bank in order to spread current peaks. If no DRAM has been programmed for a bank then no transitions occur on the relevant RAS or CAS strobes and all unused RAS and CAS strobes (i.e. strobes not used due to the choice of bank/byte mode and sub-banks) will remain inactive during a refresh cycle.

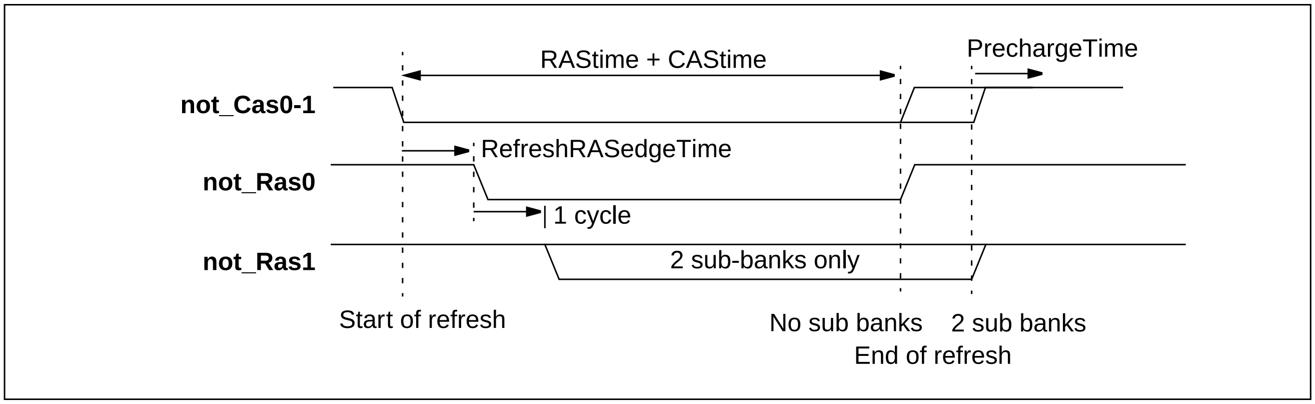


Figure 10.4 Generic refresh access

| Name | Programmable value |
|---|---|
| PrechargeTime | 1 - 8 cycles |
| RefreshInterval | (1 - 16) × 128 cycles |
| RefreshRASedgeTime | 1 or 2 cycles after start of refresh |

Table 10.4 Refresh parameters

The EMI ensures that **not_Cas** and **not_Ras** are both high for the required time before every refresh cycle by inserting a **PrechargeTime** in the last bank being accessed and ensuring all **PrechargeTimes** are complete**.**

No refreshes will take place until 1 is written to the **EMIDRAMinitialize** register.

### 10.1.3 SRAM or peripheral access cycles

A generic SRAM/peripheral type of access is provided, which is suitable for direct interfacing to a wide variety of SRAM, ROM, EPROM, Flash and peripheral devices. No internal sub-decoding is provided with banks in this configuration. All of the named times shown in Figure 10.5 together with other parameters such as bank size and bank size dependent shifts are programmable to suit a wide variety of device types. For details of the configuration of the EMI see section 10.2.

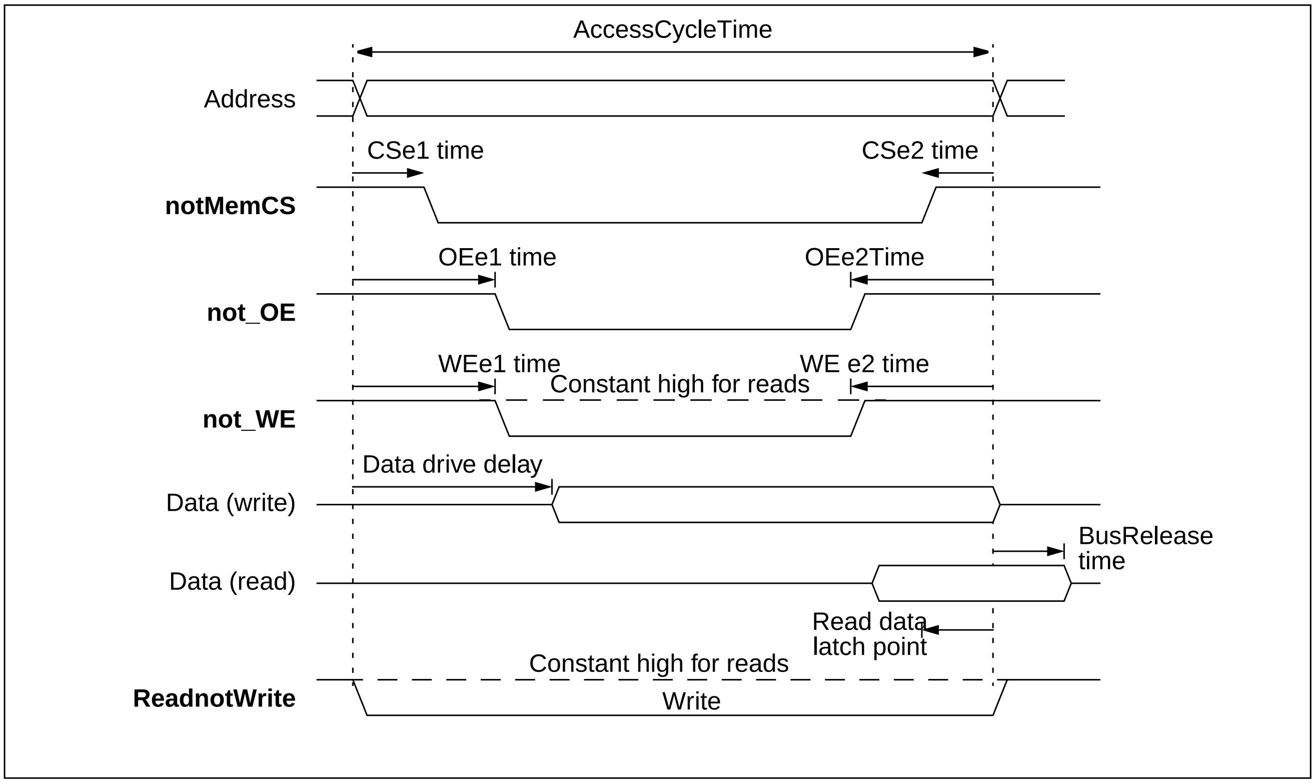


Figure 10.5 Generic peripheral access

### 10.1.4 Wait

**MemWait** is provided so that external cycles can be extended to enable variable access times, for example, shared memory access. **MemWait** is only effective during accesses to SRAM / peripheral banks and is ignored during accesses to DRAM banks. The STi5500 can only accept synchronous **MemWait** signals. Synchronous **MemWait** allows wait states to be inserted at precise times during the access. The **MemWait** signal can be enabled on a per bank basis.

**MemWait** has the effect of freezing the state of the strobes for the duration of the cycles in which it was sampled high. Any strobe transitions occurring on the sampling edge or the falling edge immediately after this will not be inhibited. However transitions on the rising and falling edges of the following cycle will not occur. Figure 9.4 and Figure 9.5 show the extension of the external memory cycle and the delaying of strobe transitions.

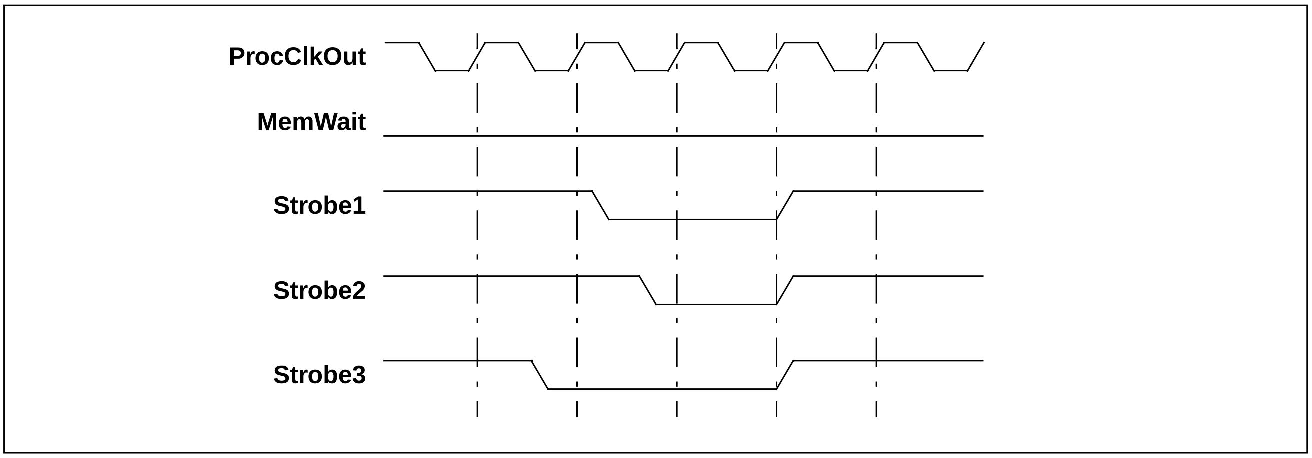


Figure 10.6 Strobe activity without **MemWait**

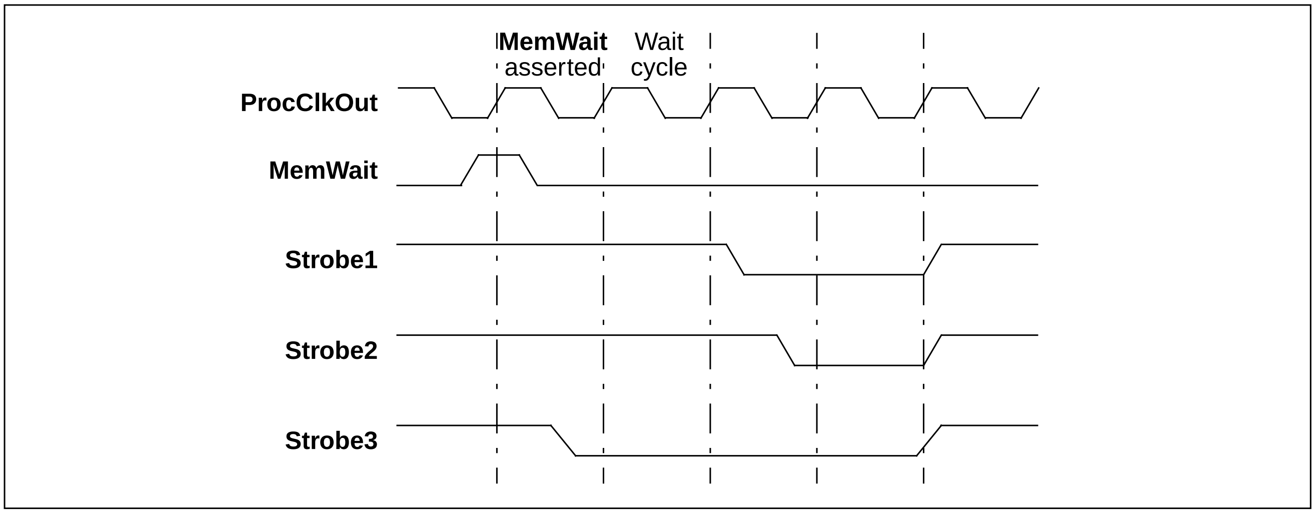


Figure 10.7 Strobe activity with **MemWait**

## 10.2 EMI Configuration

The EMI configuration is held in memory-mapped registers. The function of the registers is to eliminate external decode and timing logic. Each EMI bank has several parameters which can be configured. The parameters define the structure of the external address space and how it is allocated to the four banks and the timing of the strobe edges for the four banks.

Each EMI bank has 64 bits of configuration data which is held in four 16-bit configuration registers In addition there is a **EMIConfigLock** register for each bank, a **EMIConfigStatus** register and a **EMIDRAMInitialize** register. For safe configuration, each of the four banks should be configured after reset and then have their configuration locked by writing to the **EMIConfigLock** register before any access to an external bank is made.

# 11 System services

System services includes all the necessary logic to initialize and sustain operation of the device and also provides support for diagnostic facilities. Alternative diagnostic facilities may be available through the Diagnostic Control Unit (DCU); please consult your local STMicroelectronics field application engineer if you require assistance.

## 11.1 Reset and initialization

The STi5500 has a **notRST** pin which initializes the entire chip to a known state when asserted (low) for a period during or after power-up.

**notRST** initializes the device and causes it to enter its boot sequence unless overridden by the Diagnostic Control Unit. **notRST** must be asserted (low) at power-on, and may be asserted at other times.

When **notRST** is asserted (low), all modules are forced into their power-on reset condition. The clocks are stopped. When **notRST** is de-asserted, internal logic allows the clocks to stabilize before the chip begins its initialization sequence.

## 11.2 Debug

To select OS-Link operation, hold the pin **Brm0/Oslink_Sel** low via a resistor during reset. To select DCU operation, hold the pin **Brm0/Oslink_Sel** high via a resistor during reset.

### 11.2.1 Pinout

During OS-Link operation, **PIO3<0-4>** have these re-assigned functions:

| Standard (PIO) function | Re-assigned function |
|---|---|
| PIO3<0> | OS-Link data in |
| PIO3<1> | OS-Link data out |
| PIO3<2> | CPUReset |
| PIO3<3> | CPUAnalyse |
| PIO3<4> | ErrorOut |

Figure 11.1 Pins used during OS-Link operation

A 1kΩ pull-down resistor is needed on the **ErrorOut** (**PIO3**<**4**>) pin for correct OS-Link operation.

### 11.2.2 CPUReset

**CPUReset** is provided as a functional reset which is quicker to reboot as the PLL is not reset. In other respects the effect is the same as **notRST**.

### 11.2.3 CPUAnalyse

If **CPUAnalyse** is taken high when the STi5500 is running, the STi5500 will halt at the next descheduling point. **CPUReset** may then be asserted. When **CPUReset** comes low again the STi5500 will be in its reset state, and information on the state of the machine when it was halted by the assertion of **CPUAnalyse**, is maintained permitting analysis of the halted machine.

An input link will continue with outstanding transfers. An output link will not make another access to memory for data but will transmit only those bytes already in the link buffer. Providing there is no delay in link acknowledgement, the link will be inactive within a few microseconds of the STi5500 halting.

If **CPUAnalyse** is taken low without **CPUReset** going high the processor state and operation are undefined.

### 11.2.4 Errors

Software errors, such as arithmetic overflow or array bounds violation, can cause an error flag to be set. This flag is directly connected to the **ErrorOut** pin. The STi5500 can be set to ignore the error flag in order to optimize the performance of a proven program. If error checks are removed any unexpected error then occurring will have an arbitrary undefined effect. The STi5500 can alternatively be set to halt-on-error to prevent further corruption and allow postmortem debugging. The STi5500 also supports user-defined trap handlers.

If a high priority process pre-empts a low priority one, status of the **Error** and **HaltOnError** flags is saved for the duration of the high-priority process and restored at the conclusion of it. Status of both flags is transmitted to the high-priority process. Either flag can be altered in the process without upsetting the error status of any complex operation being carried out by the pre-empted low priority process.

In the event of a processor halting because of **HaltOnError**, the links will finish outstanding transfers before shutting down. If **CPUAnalyse** is asserted then all inputs continue but outputs will not make another access to memory for data. Memory refresh will continue to take place.

## 11.3 Bootstrap

The STi5500 can be bootstrapped from external ROM or from a link. To boot from a link, hold **Brm1/BootFromRom** low via a resistor during reset. To boot from external ROM, hold **Brm1/BootFromRom** high via a resistor during reset.

### 11.3.1 Booting from ROM

When booting from ROM, the STi5500 starts to execute code from the top two bytes in external memory, at address #7FFFFFFE which should contain a backward jump to a program in ROM.

### 11.3.2 Booting from link

When booting from a link, the STi5500 will wait for the first bootstrap message to arrive on the link. The first byte received down the link is the control byte. If the control byte is greater than 1 (i.e. 2 to 255), it is taken as the length in bytes of the boot code to be loaded down the link. The bytes following the control byte are then placed in internal memory starting at location **MemStart**. Following reception of the last byte the STi5500 will start executing code at **MemStart**. The memory space immediately above the loaded code is used as work space. A byte arriving on the bootstrapping link after the last bootstrap byte, is retained and no acknowledge is sent until a process inputs from the link.

### 11.3.3 Peek and poke

Any location in internal or external memory can be interrogated and altered when the STi5500 is waiting for a bootstrap from link.

When booting from link, if the first byte (the control byte) received down the link is greater than 1, it is taken as the length in bytes of the boot code to be loaded down the link.

If the control byte is 0 then eight more bytes are expected on the link. The first four byte word is taken as an internal or external memory address at which to *poke* (write) the second four byte word.

If the control byte is 1 the next four bytes are used as the address from which to *peek* (read) a word of data; the word is sent down the output channel of the link.

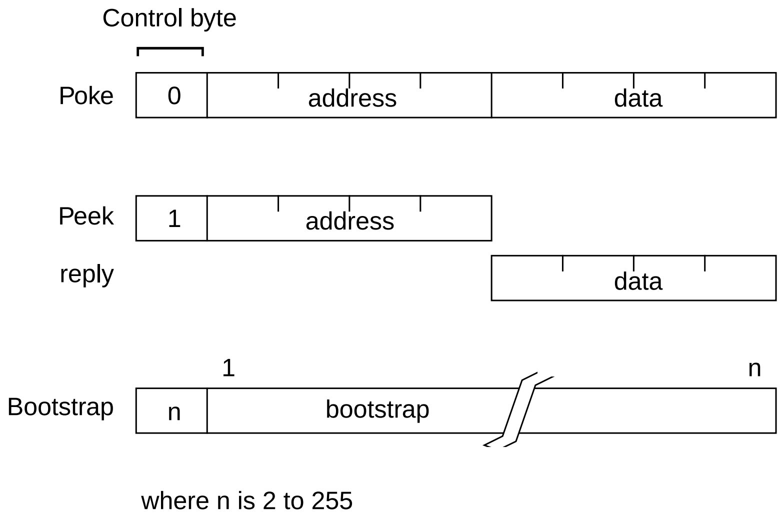


Figure 11.2 Peek, poke and bootstrap

Note, *peeks* and *pokes* in the address range #20000000 to #3FFFFFFF access the internal peripheral device registers. Therefore they can be used to configure the EMI before booting. Note that addresses that overlap the internal peripheral addresses (#20000000 to 3FFFFFFF) can not be accessed via the link.

Following a *peek* or *poke*, the STi5500 returns to its previously held state. Any number of accesses may be made in this way until the control byte is greater than 1, when the STi5500 will commence reading its bootstrap program.

# 12 Diagnostic controller

The Diagnostic Controller Unit (DCU) provides a means for booting the CPU, and for the control and monitoring of all systems on the chip, via the standard IEEE 1194.1 Test Access Port. The Test Access Port is described in Chapter 13. The DCU includes on-chip hardware with ICE (In Circuit Emulation) and LSA (Logic State Analyzer) features to facilitate verification and debugging of software running on the on-chip CPU in real time. It is an independent hardware module with a private link from the host to support real-time diagnostics.

## 12.1 Diagnostic hardware

The on-chip diagnostic controller assists in debugging, while reducing or eliminating the intrusion into the target code space, the CPU utilization, and impact on the application. As shown in Figure 12.1, the DCU and TAP provide a means of connecting a diagnostic host to a target board with a suitable JTAG port connector and interface.

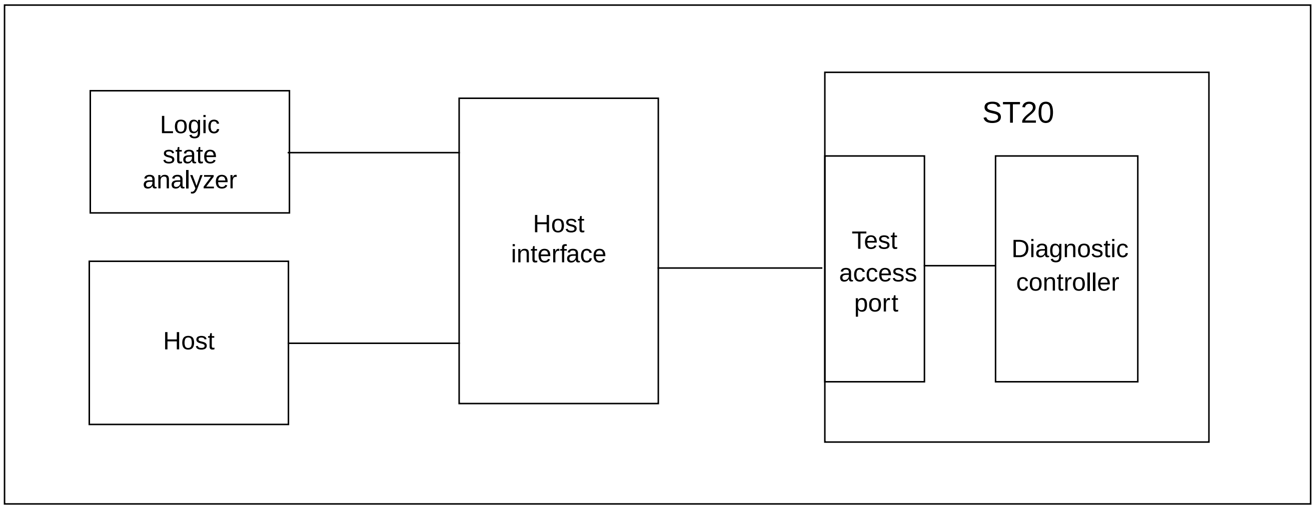


Figure 12.1 Debugging hardware

The diagnostic controller provides the following facilities for debugging from a host:

- control of target CPU and subsystems including CPU boot;
- hardware breakpoint, watchpoint, datawatch and single instruction step;
- complex trigger sequencing and choice of subsequent actions;
- non-intrusive jump trace and instruction pointer profiling;
- access to the memory of the target while the device is powered up, regardless of the state of the CPU;
- full debugging of ROM code.

When running multi-tasking code on the target, one or more processes can be single-stepped or stopped while others continue running in real time. In this case, the running threads can be interrupted by incoming hardware interrupts, with a low latency.

The host can communicate with the DCU via a private link, using the 5 standard test pins.

Target software also has access to the diagnostic facilities and access through the DCU to the host memory.

A logic state analyzer can be connected to the **TriggerIn** and **TriggerOut** pins. The response to **TriggerIn** and the events that cause a **TriggerOut** signal can be controlled by the host or by target software.

The diagnostic controller provides debugging facilities with much less impact on the software and target performance. In particular it gives:

- non-intrusive attachment to the host system;
- no intrusion into the performance of the CPU or any subsystems;
- no intrusion into the code space, so the application builder does not need to add a debugging kernel;
- no intrusion into any on-chip functional modules, including any communications facilities;
- no functional external connection pins are used.

The connections between the diagnostic controller and other on-chip modules and external hardware may vary between ST20 variants.

## 12.2 Access features

### 12.2.1 Access to target memory and peripheral registers from host

Full read and write access to the entire on-chip and external memory space and the register space is available via the TAP. This is independent of the state of the CPU.

### 12.2.2 Access from target CPU process

The CPU itself can program its own diagnostic controller. Further access may be explicitly prevented by the lock mechanism so that the application being debugged cannot interfere with the breakpoint and watchpoint settings. When the breakpoint or watchpoint match occurs, then the diagnostic controller may release the lock according to settings in the control register.

### 12.2.3 Access to host memory from target

If the target CPU accesses any address in the top half of the DCU memory space, then these accesses are mapped on to host memory via the TAP as target initiated peek and poke messages. Peek accesses and poke accesses are specifically enabled by separate property bits.

## 12.3 Software debugging features

### 12.3.1 Control of the target CPU including boot

Various state information about the target CPU may be monitored and the CPU may be controlled from the diagnostic controller via the TAP. The control of the CPU extends to stalling, forcing a trap and booting.

### 12.3.2 Non-intrusive Iptr profiling

A copy of the **Iptr** is visible as a read-only register in the diagnostic controller. This register may be read at any time. Reading this register is not intrusive on the CPU or its memory space.

### 12.3.3 Events

Support is provided by the diagnostic controller to trigger actions when certain predefined events occur.

#### Breakpoint

The function of the breakpoint is to break before the instruction is executed, but only if it really was going to be executed. A 32-bit comparator is used to compare the breakpoint register against the instruction pointer of the next instruction to be executed. The matched instruction is not executed and the CPU state, including all CPU registers, is defined as at the start of the instruction. The previous instruction is run to completion.

#### Breakpoint range

The function of a breakpoint range is equivalent to any single breakpoint but where the breakpoint address can be anywhere within a range of addresses bounded by lower and upper register values.

#### Watchpoint

The function of a watchpoint is to trigger after a memory access is made to an address within the range specified by a pair of 32-bit registers. The CPU pipeline architecture allows for the CPU to continue execution of instructions without necessarily waiting for a write access to complete. So, by the time a watchpoint violation has been detected, the CPU may have executed a number of instructions after the instruction which caused the violation. If the subsequent action is to stall the CPU or to take a hardware trap, then the last instruction executed before the stall or trap may not be the instruction which caused the violation.

#### Datawatch

The function of a datawatch is to trigger after a data value specified in one 32-bit register is written to a memory word address specified in another 32-bit register. The subsequent action is equivalent to a watchpoint.

#### Choice of subsequent actions

Following a watchpoint match, or any other condition detectable by the diagnostic controller, the subsequent action may be programmed to be one of the following:

- stall the CPU, i.e. inhibit further instructions from being executed by the CPU;
- wait until the end of the current instruction, then signal a hardware trap;
- signal an immediate hardware trap;
- continue without intrusion.

In addition, the diagnostic controller may take any combination of the following actions:

- signal on **TriggerOut** to a logic state analyzer;
- send a *triggered* message via the TAP to the host;
- unlock access by the target CPU.

### 12.3.4 Hardware single instruction step

The function of single stepping one CPU instruction is performed by using a breakpoint range over the code to be single stepped. The DCU includes a mechanism to prevent the breakpoint trap handler single-stepping itself. By selecting an inverse range, the effect of single stepping one high level instruction can be achieved.

### 12.3.5 Jump trace

Jump tracing monitors code jumps, where a jump is any change in execution flow from the stream of consecutive instructions stored in memory. A jump may be caused by a program instruction, an interrupt or a trap.

When the jump occurs, a 32-bit DCU register is loaded with the origin of the jump. This value points to the instruction which would have been executed next if the jump had not occurred. The CPU may not have completed the instruction prior to the change in flow. The diagnostic controller can be set to trace the origin of each jump, the destination, or both.

The DCU copies the details of each jump to a rolling trace buffer in memory. The trace buffer may be located in host memory, but using target memory will have less impact on performance. The tracing facility has two modes:

- Low intrusion. In this mode the DCU uses dead memory cycles to write the trace into the buffer. This means that the CPU is not delayed, but some trace information may be lost.
- Complete trace. In this mode, the CPU is stalled on every jump to ensure the data can be written to the buffer. This means that no trace information is lost, but the CPU performance is affected.

### 12.3.6 Logic state analyzer (LSA) support

Two signals, **TriggerIn** and **TriggerOut**, are provided to support diagnostics with an external LSA. The action by the DCU on receiving a **TriggerIn** signal is programmable. The selection of internal events which trigger a **TriggerOut** signal is also programmable.

### 12.3.7 Trigger combinations and sequences

Complex trigger conditions can be programmed. For example:

- the 5th time that breakpoint 3 is encountered;
- enable a watchpoint when a breakpoint occurs.

There is no software intrusion imposed by this mechanism.

## 12.4 Controlling the diagnostic controller

This section gives a summary of host communications with the diagnostic controller.

The diagnostic controller has direct access to:

- the instruction pointer,
- a selection of CPU state control signals,
- the memory bus,
- memory-mapped peripheral configuration registers.

This access does not depend on the state of the CPU. Access to non-memory-mapped peripheral configuration registers is via the CPU, and for this the CPU must be active and running the appropriate handler.

The host can give two commands to the diagnostic controller: *peek* and *poke*. *Peek* reads memory locations or configuration registers, and *poke* writes to memory locations or configuration registers. The diagnostic controller responds to a *peek* command with a *peeked* message, giving the contents of the peeked addresses.

The diagnostic controller has registers, which are accessed from the host using *peek* and *poke* commands. The registers are used to control breakpoints, watchpoints, datawatch, tracing and other facilities.

The target CPU can also access these registers using the normal load and store instructions, so the target software running on the CPU can program its own diagnostic controller. A lock is provided to prevent CPU access, which can be released by the diagnostic controller when a breakpoint or watchpoint match occurs.

In addition, the target CPU can *peek* and *poke* the host via the diagnostic controller by reading or writing addresses in the top half of the memory space of the diagnostic controller. This facility can be disabled.

Various different types of CPU events can be selected as *trigger events*. When an trigger event occurs, the diagnostic controller can send a *triggered* message.

The four types of message are summarized in Table 12.1. The messages are distinguished by the two least significant bits of the message header byte.

| Message type | Direction | Bit 1 | Bit 0 | Meaning |
|---|---|---|---|---|
| *poke* | Command. | 0 | 0 | Write to one or more addresses. |
| *peek* | Command. | 0 | 1 | Read from one or more addresses. |
| *peeked* | Opposite to *peek* command. | 1 | 0 | The result of a *peek* command. |
| *triggered* | DCU to host. | 1 | 1 | A trigger event has occurred. |

Table 12.1 Types of diagnostic controller message

Messages may be initiated from either the host or the target. Target initiated messages, which constitute asynchronous or unsolicited messages, can be enabled by a property bit.

Messages are composed of a header byte followed by zero or more data bytes, depending on the type of message. The formats for the four message types are shown in Figure 12.2.

:

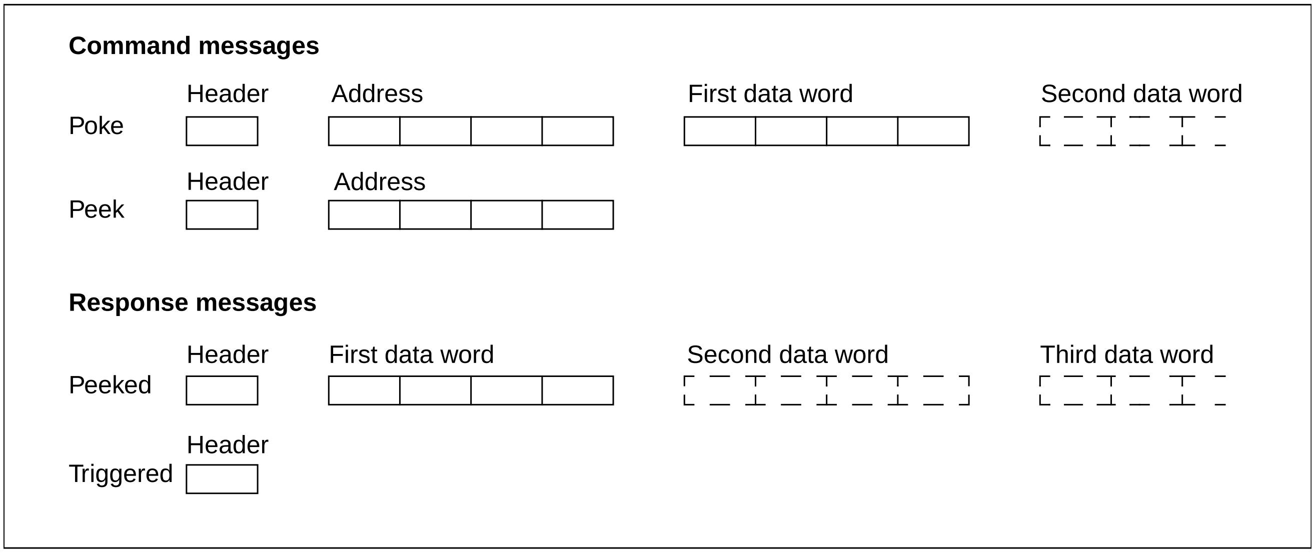


Figure 12.2 Message formats

## 12.5 Peeking and poking the host from the target

The target CPU can *peek* and *poke* the host via the diagnostic controller. This is done by reading or writing a single word to a block of addresses within the DCU register block. The DCU will then send a *peek* or *poke* message to the host. After a host *peek*, the target CPU will wait until the host responds with a *peeked* message, which the DCU returns to the CPU as memory read data.

Peeking and poking the host from the target can be enabled or disabled. After reset, these bits are cleared, so peek and poke from the target are disabled.

# 13 Test access port

The STi5500 Test Access Port (TAP) conforms to IEEE standard 1149.1 in all respects except that there is no Boundary Scan register.

The TAP consists of five pins: **Tms**, **Tck**, **Tdi**, **Tdo** and **notTrst**. **Tdo** can be overdriven to the power rails, and **Tck** can be stopped in either logic state.

The instruction register is 5 bits long, with no parity, and the pattern “00001” is loaded into the register during the *Capture-IR* state.

There are two defined public instructions, see Table 13.1. All other instruction codes are reserved.

| **Instruction code [a]** | | | | | **Instruction** | **Selected register** |
|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 1 | 0 | IDCODE | Identification |
| 1 | 1 | 1 | 1 | 1 | BYPASS | Bypass |

Table 13.1 Instruction codes

[a] MSB ... LSB; LSB closest to **Tdo.**

There are two test data registers; **Bypass** and **Identification**. These registers operate according to 1149.1.

The identification code for revisions DA, DB etc. is #M51D9041, see Table 13.2.

| bit 31 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | bit 0 [a] |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Mask rev** | | | | **b** | **ST20 family** | | | | | **Variant** | | | | | | | | | | **STMicroelectronics manufacturers id** | | | | | | | | | | | **c** |
| 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 1 | 1 | 1 | 0 | 1 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 1 |
| M | | | | 5 | | | | 1 | | | | D | | | | 9 | | | | 0 | | | | 4 | | | | 1 | | | |

Table 13.2 Identification code

[a] Closest to **TDO.**

[b] 0 indicates STMicroelectronics part, 1 indicates customer part.

[c] Defined as 1 in IEEE 1149.1 standard.

# 14 Serial link interface (OS-Link)

The STi5500 has an OS-Link based serial communications subsystem. The OS-Link is used to provide serial data transfer and its main function is for booting the device and debugging during software development.

The OS-Link is a serial communications engine consisting of two signal wires, one in each direction. OS-Links use an asynchronous bit-serial (byte-stream) protocol, each bit received is sampled five times, hence the term *oversampled links* (OS-Links). The OS-Link provides a pair of channels, one input and one output channel.

The OS-Link is used for the following purposes:

- Bootstrapping - the program which is executed at power up or after reset can reside in ROM in the address space, or can be loaded via the OS-Link directly into memory.
- Diagnostics - diagnostic and debug software can be downloaded over the link connected to a PC or other diagnostic equipment, and the system performance and functionality can be monitored.

## 14.1 OS-Link protocol

The quiescent state of a link output is low. Each data byte is transmitted as a high start bit followed by a one bit followed by eight data bits followed by a low stop bit (see Figure 14.1). The least significant bit of data is transmitted first. After transmitting a data byte the sender waits for the acknowledge, which consists of a high start bit followed by a zero bit. The acknowledge signifies both that a process was able to receive the acknowledged data byte and that the receiving link is able to receive another byte. The sending link reschedules the sending process only after the acknowledge for the final byte of the message has been received. The link allows an acknowledge to be sent before the data has been fully received.

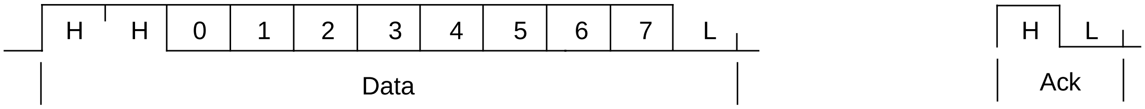


Figure 14.1 OS-Link data and acknowledge formats

## 14.2 OS-Link speed

The OS-Link data rate is 19.98 Mbits/s, but it will operate correctly when connected to 20 Mbits/s OS-Links.

## 14.3 OS-Link connections

Links are TTL compatible and intended to be used in electrically quiet environments, between devices on a single printed circuit board or between two boards via a backplane. Direct connection may be made between devices separated by a distance of less than 300 mm.

For longer distances a matched 100 ohm transmission line should be used with series matching resistors (RM), see Figure 14.3. The value of RM to match a 100Ω transmission line is 75Ω. When this is done the line delay is less than 0.4 bit time to ensure that the reflection returns before the next data bit is sent.

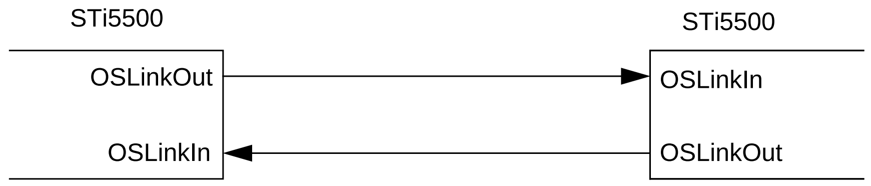


Figure 14.2 OS-Links directly connected

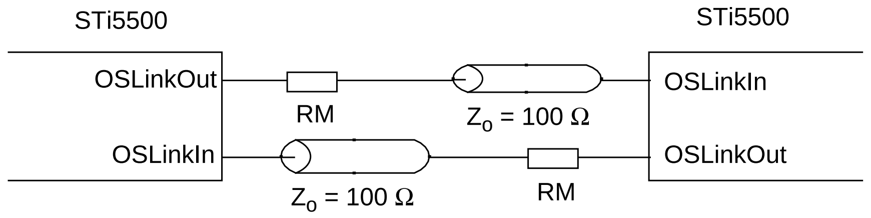


Figure 14.3 OS-Links connected by transmission line

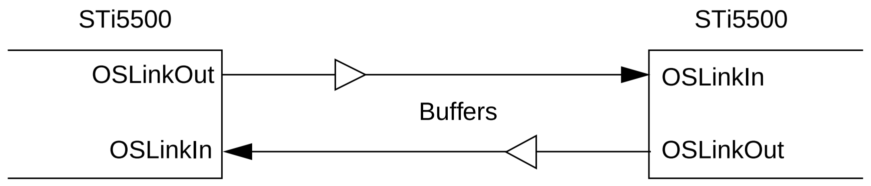


Figure 14.4 OS-Links connected by buffers

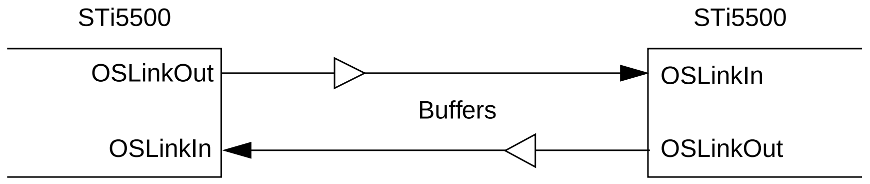


Figure 14.5 OS-Links connected by buffersOS-Links connected by buffers

# Part C Video and audio

# 15 Data flow

This chapter describes the normal data flow through the STi5500 from the incoming transport stream to the outgoing analog video and PCM audio. It shows how the picture and sound modules of the part are used together. The individual modules are described in the appropriate chapters.

## 15.1 On-chip modules

The STi5500 reads in an MPEG-2 transport stream, demultiplexes it, decodes the audio and video elementary streams and creates a video picture and audio PCM.

Demultiplexing extracts video and audio MPEG streams plus other PES data such as teletext and DVB subtitles. Hardware modules are provided on-chip for decoding the MPEG video and audio. The data before decoding is called compressed data (CD), and digital video data after decoding is called pixel data.

The on-chip modules processing the compressed data streams from the incoming transport stream to the decoders are shown in Figure 15.1.

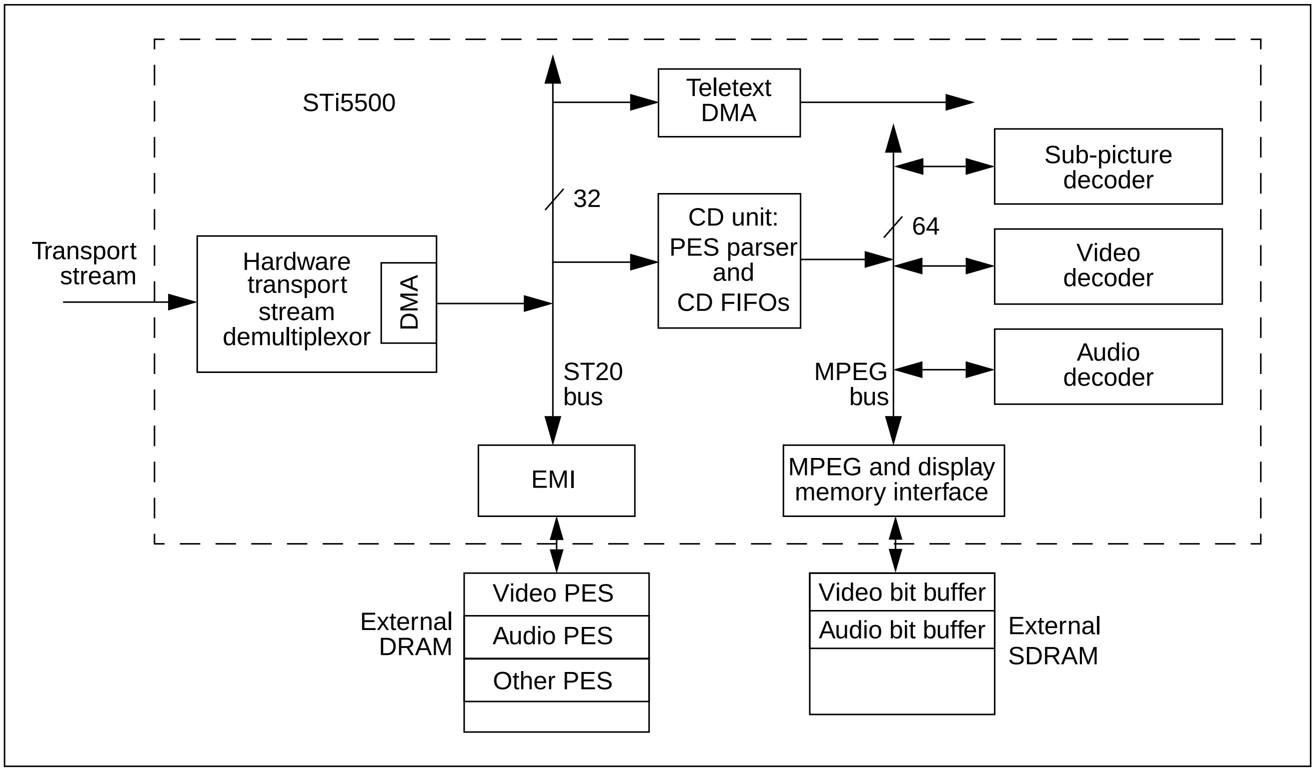


Figure 15.1 Compressed data modules

The on-chip modules processing the decoded data from the decoders to the video and audio output are shown in Figure 15.2.

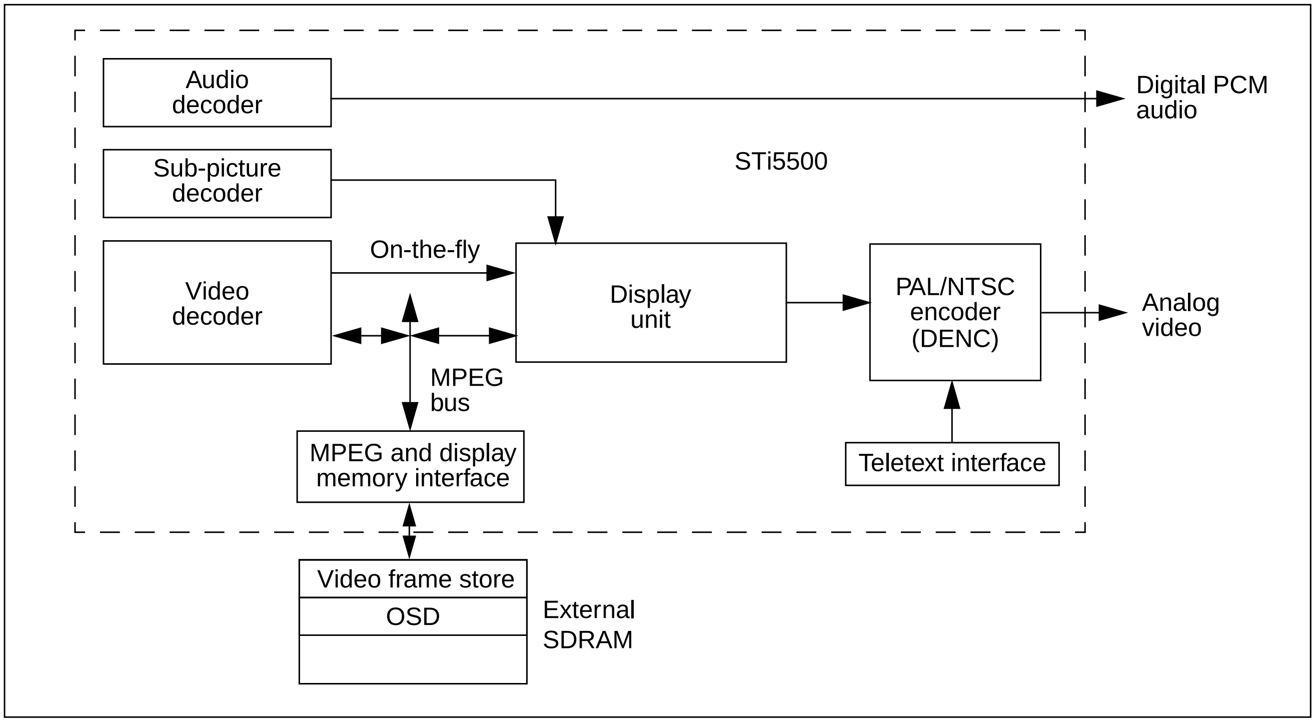


Figure 15.2 Decoded data modules

## 15.2 Video data flow

The data flow for MPEG-2 video streams is summarized in Figure 15.3. Rectangular boxes represent processing modules, and rounded boxes represent buffers and FIFOs.

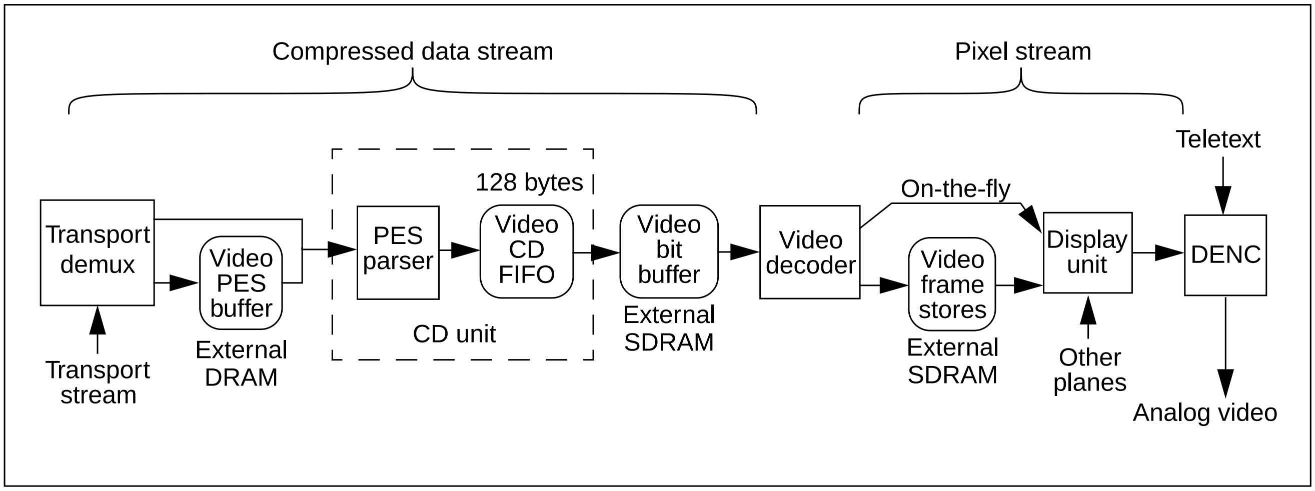


Figure 15.3 MPEG-2 video data flow

The compressed data (CD) is read in as a transport stream by the hardware transport stream demultiplexor, as described by Chapter 16. This module demultiplexes the stream, extracting the required packets.

Any selected video stream PES packets are sent on by the transport demultiplexor DMA engine. The stream can be either:

- written into a circular video PES buffer in external DRAM by the transport demultiplexor DMA and then read into the CD unit by an MPEG DMA or
- sent directly to the CD unit by the transport demultiplexor DMA.

The high speed MPEG DMA engines are described in Chapter 17. Like the transport demultiplexor DMA, these DMA engines require very little intervention by the CPU.

Video data enters the CD unit through the PES parser, which passes the data to the video CD FIFO. The video CD FIFO holds 128 bytes and writes 512-bit bursts into the video bit buffer in external SDRAM.

The video decoder (described in Chapter 18) reads 1024-bit bursts from the video bit buffer. It decodes the compressed bit stream and produces a pixel stream. I-frames and P-frames must be written into video frame stores, while B-frames may be either written into a frame store or sent 'on-the-fly' directly to the display unit.

The display unit is described in Chapter 21. It converts the blocks of pixels into rows and performs filtering and pan/scan. It then mixes the video with the other display planes and sends a pixel stream to the on-chip PAL/NTSC encoder (DENC).

The DENC is described in Chapter 23. It converts the pixel streams into analog signals for output from the device. Teletext can be inserted into the output signals.

## 15.3 Audio data flow

The data flow for audio streams is summarized in Figure 15.4. Rectangular boxes represent processing modules, and rounded boxes represent buffers and FIFOs.

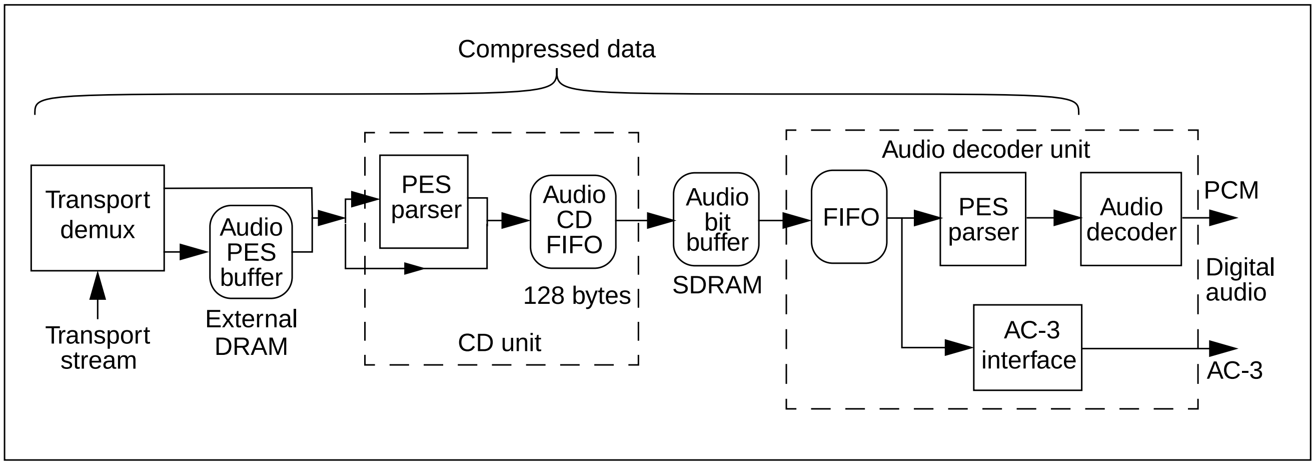


Figure 15.4 Audio data flow

As for video, the compressed audio data is read in as a transport stream by the hardware transport stream demultiplexor. This module demultiplexes the stream and extracts the required audio PES packets, which are sent on by the transport demultiplexor DMA engine. The stream can be either:

- written into a circular audio PES buffer in external DRAM by the transport demultiplexor

DMA and then read into the CD unit by an MPEG DMA or

- sent directly to the CD unit by the transport demultiplexor DMA.

If the audio data is in the form of MPEG-2 packetized PES, then the data must be sent directly to the audio CD FIFO, in which case it will not pass through the PES parser in the CD unit. Otherwise, whole PES audio data interleaved with whole PES video data should be sent on the video route through the PES parser. The PES parser will separate out any audio packets and route them to the audio CD FIFO. The audio CD FIFO writes 512-bit bursts into the audio bit buffer in external SDRAM. The CD unit is described in Chapter 18.

The audio decoder (described in Chapter 20) unit includes its own FIFO and PES parser. It reads from the bit buffer into its FIFO, and then the data may be either:

- passed to the PES parser, which passes the data on to the audio decoder for decoding into PCM data for output or
- passed to the AC-3 interface to send to an external Dolby AC-3 decoder.

# 16 Hardware transport stream demultiplexor

## 16.1 Introduction

The Hardware Transport Stream Demultiplexor for the STi5500 chip accepts as input a constrained DVB or DSS transport stream or a DVD program stream. It extracts from the stream the (possibly scrambled) compressed data bytes of a set of Packet Elementary Streams (PES) belonging to one user-selected program to be decoded and presented.

In addition, *service* data bytes from Section Streams are extracted from the bit stream and stored in appropriate buffers to be used by the decoder control unit.

A high speed digital interface allows transfer of transport packets between the Integrated Receiver Decoder box (IRD) and external units, either for recording or playback purposes.

This Simplified Digital Audio Video (SDAV) interface provides full support for an external IEEE1394 Link IC.

A National Renewable Security System (NRSS) interface is also integrated into the Transport Stream Demultiplexor to allow external descrambling for DSS, DVB, or DVD packets.

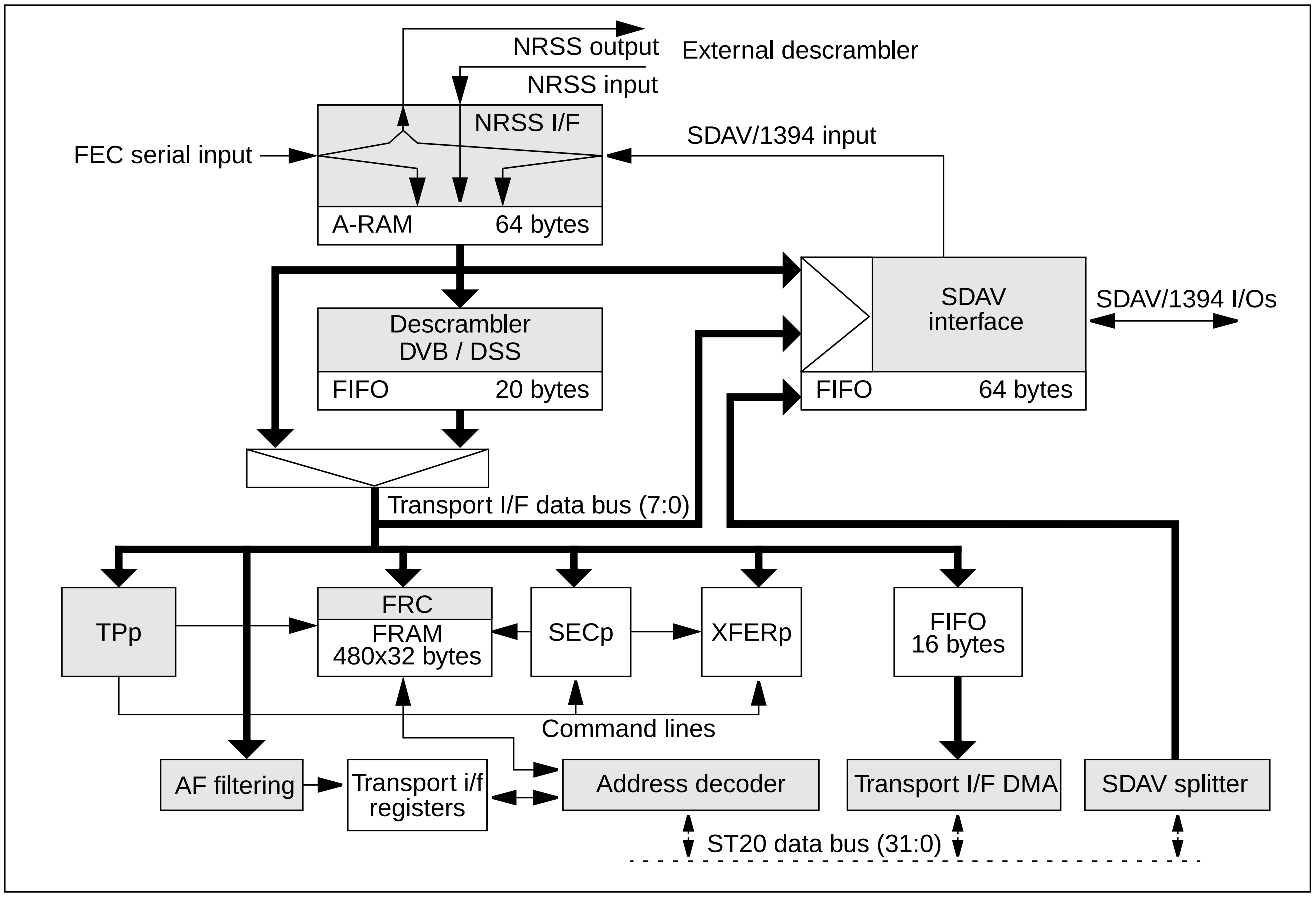


Figure 16.1 Hardware transport stream demultiplexor block diagram

### 16.1.1 MPEG-2 and DSS systems layers

Two layers are defined:

- PES packets (or Sections for Program Specific Information (PSI)) layer.
- Transport Packets (TP) layer.

The Transport Stream Demultiplexor performs complete processing at the TP layer and possibly at the PES or Section layer, in accordance with Table 16.1.

| Function | Layer DVB | Layer DSS |
|---|---|---|
| **Acquisition** | TP | TP |
| **Descrambling** | TP or PES | TP |
| **H/W Filtering**<br>**(PSI for DVB, CA for DSS)** | Section | TP |

Table 16.1 PES and Section layers

### 16.1.2 System general description

The Transport Stream Demultiplexor for the STi5500 chip is used to interface the MPEG decoders and the CPU with the incoming data stream. It is composed of the following units:

- Acquisition RAM (AR) and NRSS interface;
- Descramblers (DESCR);
- SDAV-1394 interface;
- Filter RAM (FRAM);
- Processor units:
  - transport processor;
  - section processor;
  - transfer processor;
- Adaptation field filtering;
- Clock recovery;
- DMA engine.

### 16.1.3 Input interface (NRSS interface and acquisition RAM)

Signals at the FEC interface and Transport Stream Demultiplexor system clock are asynchronous.

TP bytes, provided by the serial-parallel converter, are buffered in the Acquisition RAM (AR), which is a FIFO memory.

The processing of a packet has to be started before a programmable level of the AR is reached, and at least a few bytes before the AR is full.

### 16.1.4 Descrambling

Both DVB and DES descramblers are implemented. For DVB, TP and PES level descrambling are supported.

For DSS the scrambling is only done at TP level. Up to 8 different key sets can be used to descramble up to 32 streams. The keys for descrambling are located in the FRAM. They are automatically loaded after the PID filtering.

If the payload in an acquired TP is scrambled, the descrambler is set up to handle descrambling and to return descrambled bytes. If the payload is not scrambled, payload bytes are sent directly.

### 16.1.5 SDAV interface

A high speed bidirectional digital interface is used to transfer TP between the IRD and an external unit.

The SDAV bus is a point-to-point connection. It only allows one source on any bus segment at a time. As the input is 40MHz and the rate on the SDAV bus is 49.1Mbit/s, some buffering is required not to run out of data.

The IEEE 1394 standard provides a single I/O interface with a simple connector that can handle numerous devices through a single port. It allows simultaneous transmissions at data rates up to 400Mbit/s.

Because of the complexity of the IEEE 1394 standard and the cost of its implementation, the STi5500 includes a single SDAV interface which provides full support for an external IEEE 1394. This block takes an incoming packet stream and reformats it for the SDAV/1394 bus. It also takes incoming SDAV/1394 bus information and regenerates the corresponding packet stream.

A dedicated MPEG DMA engine is also provided, as described in Chapter 17.

SDAV or IEEE 1394 mode, input or output mode are selected by software.

### 16.1.6 PID filtering

This block contains a filter to receive the TPs of only one program. It extracts the transport packets of up to 32 streams from the incoming bit stream.

### 16.1.7 Section filtering

A second filter function is applied to all section type data. For each stream there can be up to 32 targets to which the incoming section header can be compared. The maximum length of the targets is 16 bytes for DSS and 14 bytes for DVB. Each bit of each target can be masked individually. For one target byte, two bytes of RAM are needed. The total number of target bytes is defined by the size of the filter RAM array that is used.

### 16.1.8 Adaptation field filtering

Filtering is performed to extract PCR informations or to discard any undesired data contained in the extracted TP.

## 16.2 Detailed description

### 16.2.1 Input interface (NRSS interface and acquisition RAM)

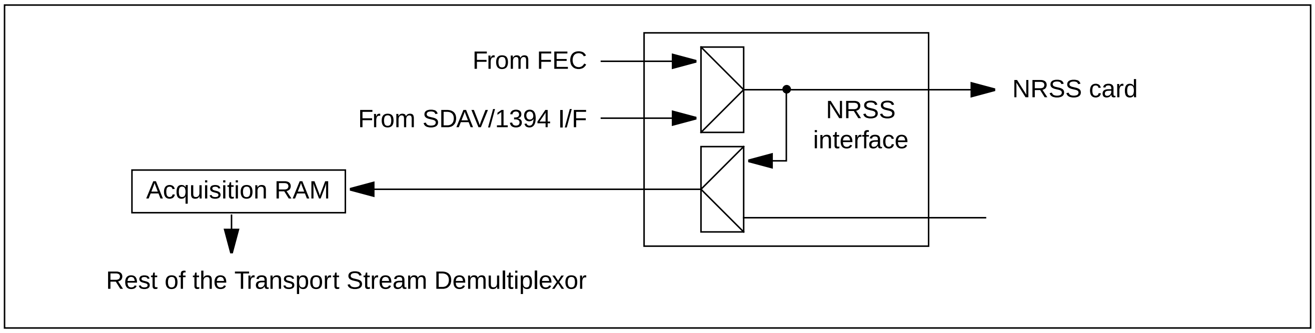


Figure 16.2 Input interface

The Transport Stream Demultiplexor receives the TP through the input interface section; it is a fully asynchronous FIFO buffer (64 bytes) that uncouples read and write clocks.

Data are latched on the falling edge of **F_B_Clk**. The four signals of this interface have different names and meanings in different modes. They are always inputs. Table 16.2 gives the signal names for the different modes.

| Type | DVB/DSS | DVD |
|---|---|---|
| I | **F_Data** | |
| I | **F_B_Clk** | |
| I | **F_P_Clk** | **F_D_Valid** |
| I | **F_Error** | **F_P_Start** |

Table 16.2 Input interface signal names

### DVB/DSS mode

The **F_P_Clk** is active high during the significant bits of the packet (188 x 8 for DVB, 130 x 8 for DSS). On this pin, the rising edge is detected and the internal FIFO counter is reset.

The **F_Error** signal should be active high for an entire packet if there is an error somewhere in the packet. These packets will not be written into the AR. This signal should only change value at the rising edge of **F_P_Clk**.

### DVD mode

With the **F_P_Start** signal, active during the first bit of the first byte of a packet, the Transport Stream Demultiplexor detects the beginning of a packet.

On this pin, the rising edge is detected and the internal FIFO counter is reset.

The **F_D_Valid** signal will be active during a burst transmission (data are transferred in a burst of at least 8 bits long).

There can be a gap of one or more clock cycles between bytes but no gaps within the 8 bits of a byte.

After a rising edge transition on **F_P_Start** and after serial/parallel conversion (MSB first), data bytes are latched into the FIFO when **F_D_Valid** is high (see Figure 16.3).

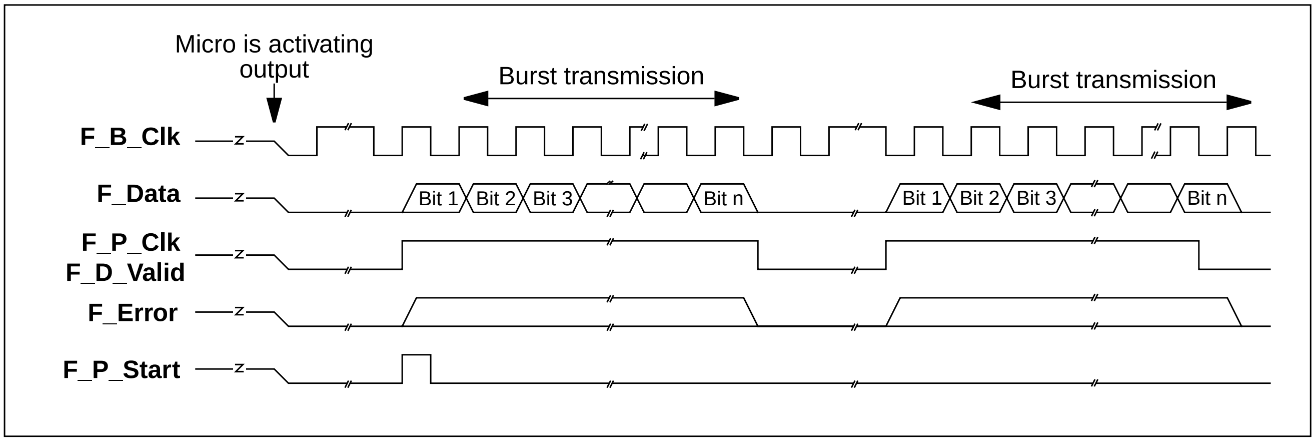


Figure 16.3 Serial input interface from Channel IC or Link IC

### Input and output data rates

The input interface is a serial interface with a maximum peak data of 60Mbit/s (from the Link IC) or 60.54Mbit/s (from the Channel IC).

The Transport Stream Demultiplexor block is specified at 60Mbit/s. The maximum input data rate is 59.5Mbit/s or 7.43Mbytes/s (7/8 of 68Mbit/s).

| Clocks | Description | Value |
|---|---|---|
| **SYS_CLK** | Descrambler clock | 60MHz |
| **F_B_Clk** | Bit clock signal | up to 59.5MHz |

Table 16.3 Data rates

The maximum output data rate is 15 Mbyte/s.

### NRSS interface

NRSS can be used when input is FEC or SDAV (see Figure 16.1, Figure 16.3 and Figure 16.4). The NRSS interface makes it possible to descramble the channel off chip.

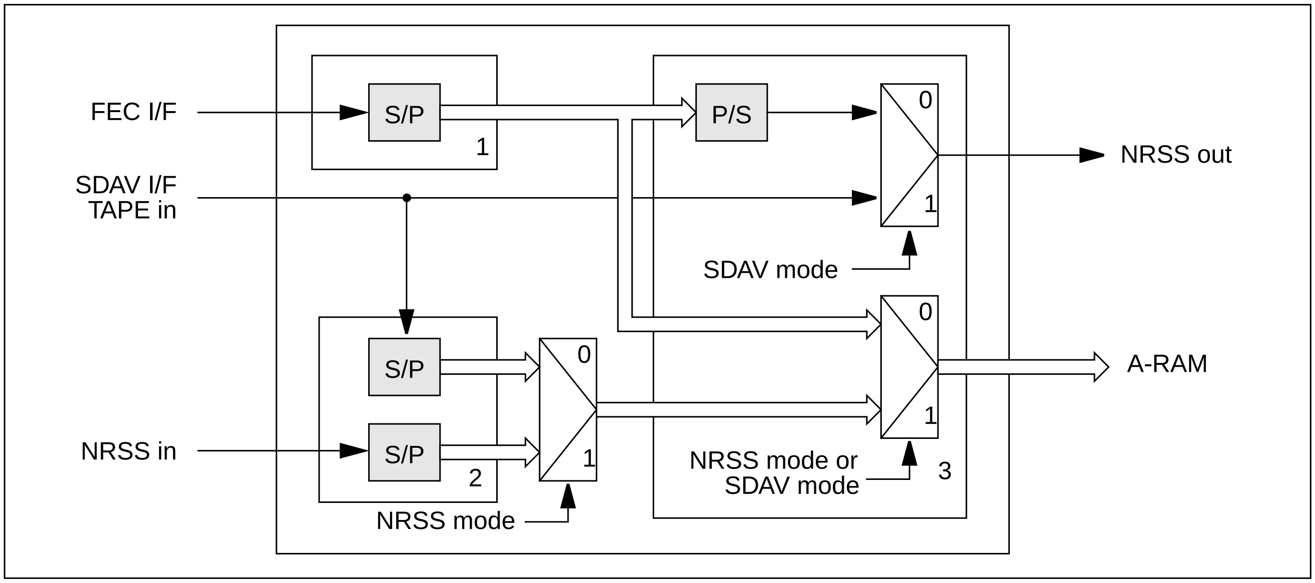


Figure 16.4 Input interface details

The existing paths are the following:

FEC → AR
FEC → NRSS → AR
SDAV → AR
SDAV → NRSS → AR

Two separate mux controls are used: one to control the top mux that selects if the input is coming from the FEC interface or the SDAV input and a second signal to control whether NRSS is used or not. The data can pass through without going out to the NRSS. The other FEC signals pass through this block to make sure that proper timing is maintained.

Serial to parallel conversion is performed after NRSS (see Figure 16.4). The **BIT_ORDER** register is used to select MSB or LSB first in the serial-parallel converter.

When input is FEC, the **Nrss_Clk** coming from the Transport Stream Demultiplexor is discontinuous. Consequently the data has to be maintained at the end of each byte to work properly.

When input is SDAV, the **Nrss_Clk** is discontinuous. The SDAV block synchronizes the incoming data again to the **Sys_Clk** and sends the data with a **F_B_Clk** and **F_D_Valid** signal (if the data was already at the **Sys_Clk** rate then the **Nrss_Clk** should be continuous).

In both cases, the inputs to the NRSS block are shifted from a serial bit stream into a serial to parallel converter shift register using the incoming clock. In the case of the FEC, this is **F_B_Clk** (FEC clock) and in the case of the SDAV it is the SDAV clock (usually 49.152MHz). The parallel byte is then loaded into a register and a single bit is generated that toggles on each new byte. That signal is then sampled with the **Sys_Clk**. This asynchronous sampling takes a couple of clock cycles. The byte is then loaded into the output shift register for the NRSS interface and is shifted out using the **Sys_Clk**. If the **Sys_Clk** is faster than the incoming clock then there will be **Sys_Clk** cycles where there is no data available to shift out. When there is no available data, the **Nrss_Clk** output is forced to remain low for that clock cycle. This mechanism can be broken by making the FEC clock (**F_B_Clk**) faster than **Sys_Clk**.

Normally the AR has all four of the FEC input signals (**F_B_Clk**, **F_Data**, **F_P_Clk**, and **F_Error**). Since the NRSS interface has only clock and data there is no indication of the beginning of a packet. Within the NRSS interface the Transport Stream Demultiplexor looks for the **sync_byte** (0x47) coming from the NRSS card to indicate the beginning of a packet and uses that to generate a **packet_clock**. In DVB the **sync_byte** is already there but in DSS the Transport Stream Demultiplexor adds it to the beginning of the packet and then strips it back off when it comes back from the NRSS card.

### NRSS interface timing

Figure 16.5 and Table 16.4 show the timing in DVB/DSS mode. The data is output on the rising edge of **Nrss_Clk**, as shown in Figure 16.5.

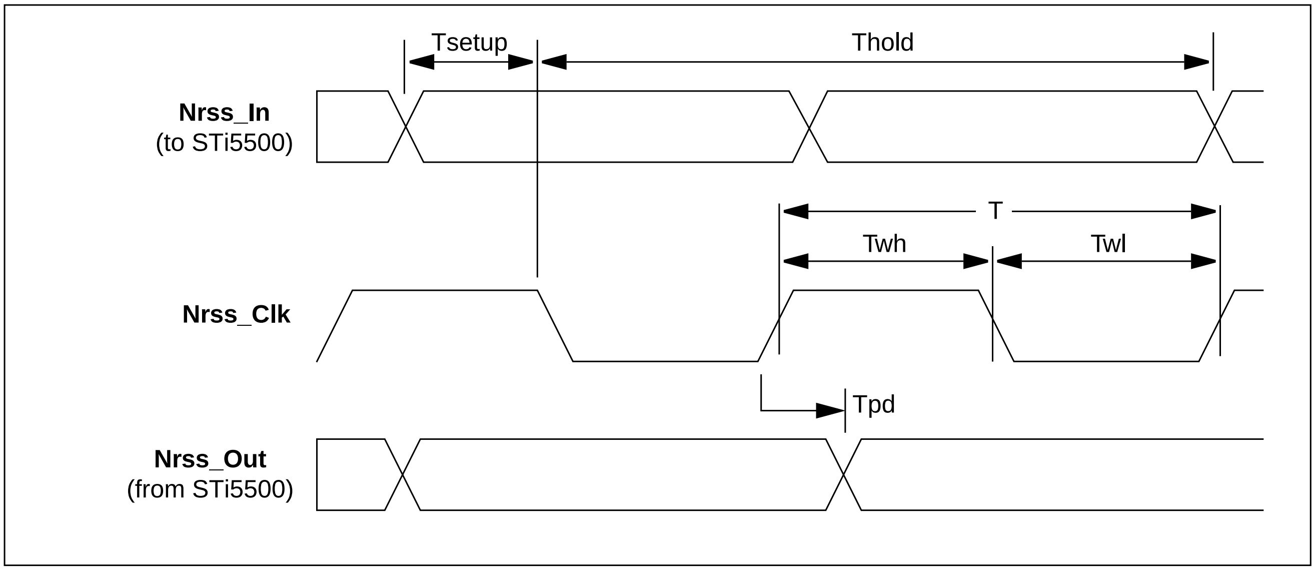


Figure 16.5 NRSS interface timing in DVB/DSS mode

| Name | Min (ns) | Max (ns) | Comment |
|---|---|---|---|
| Tpd | 0 | 3 | Propagation delay |
| Tsetup | 5 | | Set up data to clock |
| Thold | 2 | | Hold clock to data |
| Twh | 9 | | Clock width high |
| Twl | 9 | | Clock width low |
| T | 20 | | Clock period |

Table 16.4  NRSS timing (DVB and DSS)

### Size of the FIFO (AR)

The internal FIFO counter is reset by the rising edge of the **packet_clock** signal in DVB/DSS mode or by the rising edge of **sector_start** in DVD mode. When the following packet arrives, the last bytes of the packet in process have to be read. To ensure this, the upper limit of this FIFO is programmable by software so that the last byte of a packet is written as high as possible in the FIFO (see Figure 16.6).

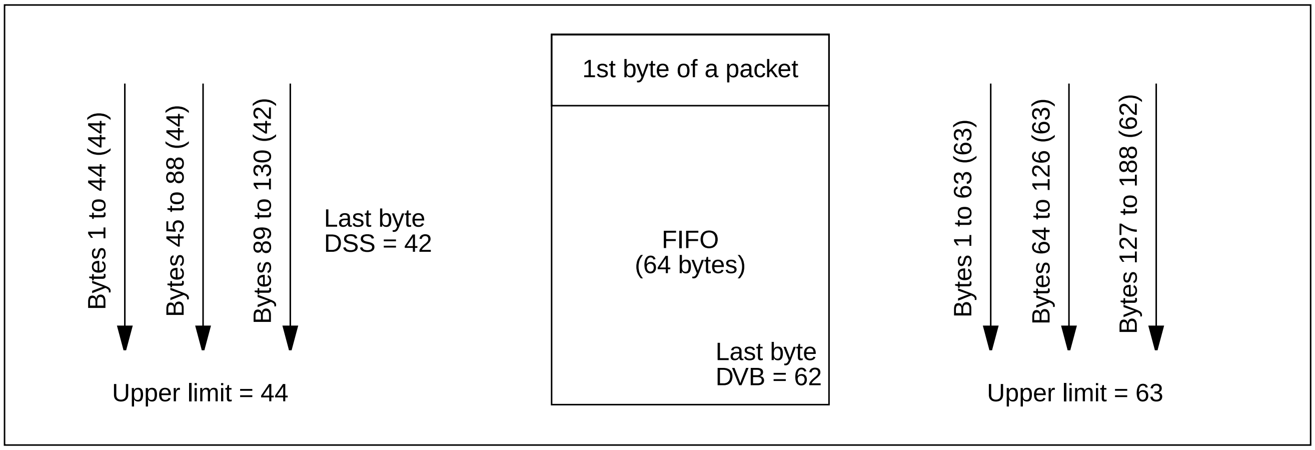


Figure 16.6 Acquisition RAM

The optimized AR size is different for each mode:

- DSS: 44 (2*44 + 42 = 130)
- DVB: 63 (2*63 + 62 = 188)
- DVD: 53 (38*53 + 52 = 2066) for DVD mode
  or 58 (43*58 + 54 = 2548) for CD mode

### 16.2.2 Descrambler

The Transport Stream Demultiplexor includes two descramblers which conform to the DVB/DES descrambler specifications.

| | Scrambling control bits | MSB | LSB (if MSB = 1) |
|---|---|---|---|
| DVB TP Level | Byte # 4<br>Bits(7:6) | 1: scrambled<br>0: non-scrambled | 1: odd key<br>0: even key |
| DVB PES Level | Byte # 11<br>Bits(5:4) | 1: scrambled<br>0: non-scrambled | 1: odd key<br>0: even key |
| DSS | Byte # 1<br>Bits(5:4) | 1: non-scrambled<br>0: scrambled | 1: odd key<br>0: even key |

Table 16.5

Note: Byte # 1 = 1st byte of the TP.
Bit(7) = MSB Descrambler DVB.

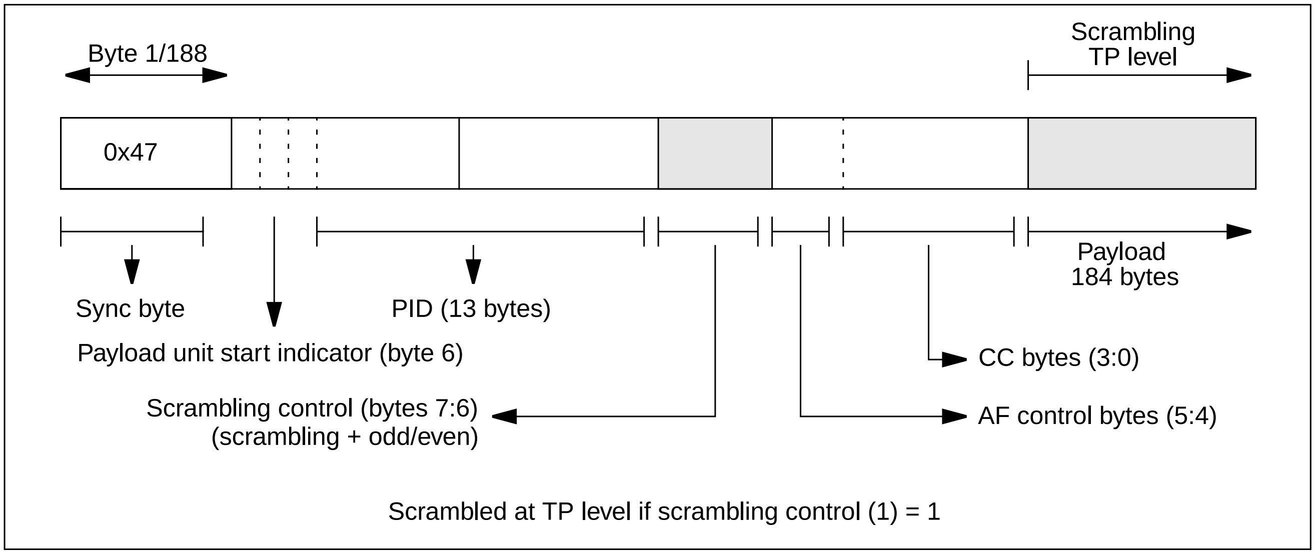


Figure 16.7 TP header

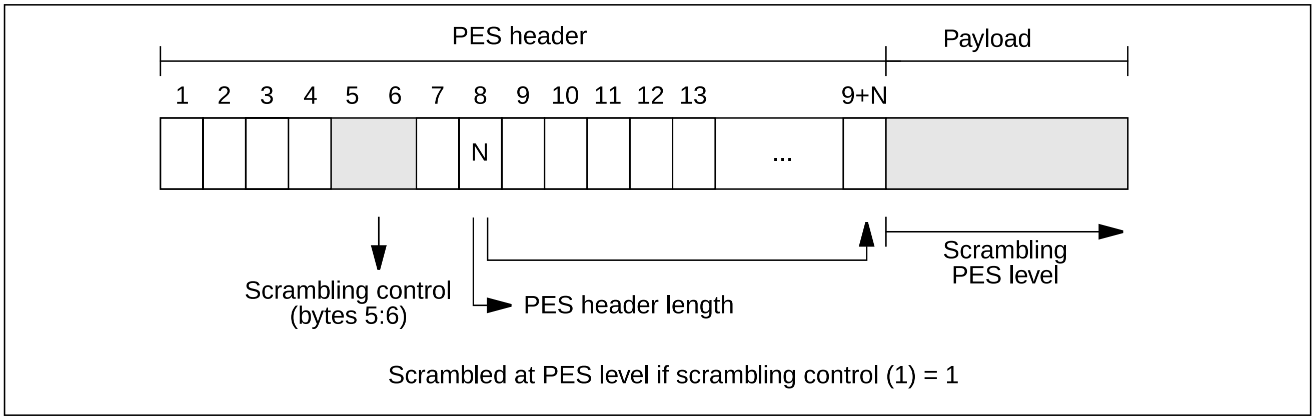


Figure 16.8 PES header

### Description

The scrambling algorithm operates on the payload of a TP in the case of TP-level scrambling. A structuring of PES packets is used to implement PES level scrambling with the same scrambling algorithm. The scrambling of MPEG-2 sections is at TP level.

### PES level scrambling

The PES scrambling method requires that the PES packet header shall not be scrambled (ISO/IEC 13818-1).

The DVB standardization requires the following, as shown in Figure 16.9:

- the header of a scrambled packet shall not span multiple TP - *Recommendation 2*
- the TP containing parts of a scrambled PES packet shall not contain an Adaptation Field (with the exception of the TP containing the end of the PES packet) - *Recommendation 3*
- the TP carrying the start of a scrambled PES packet is filled by the PES header and the first part of the PES payload - *Recommendation 3*

In this way, the first part of the PES packet payload is scrambled exactly as a TP with a similar payload. The remaining part of the PES packet payload is split in super-blocks of 184 bytes. Each block is scrambled exactly as a TP payload of 184 bytes.

- the end of the PES packet payload is aligned with the end of the TP by inserting an Adaptation Field of suitable size (as required in ISO/IEC 13818-1).

If the length of a packet is not a multiple of 184 bytes, the last part of the PES packet payload (from 1 to 183) is scrambled exactly as a TP with a similar payload.

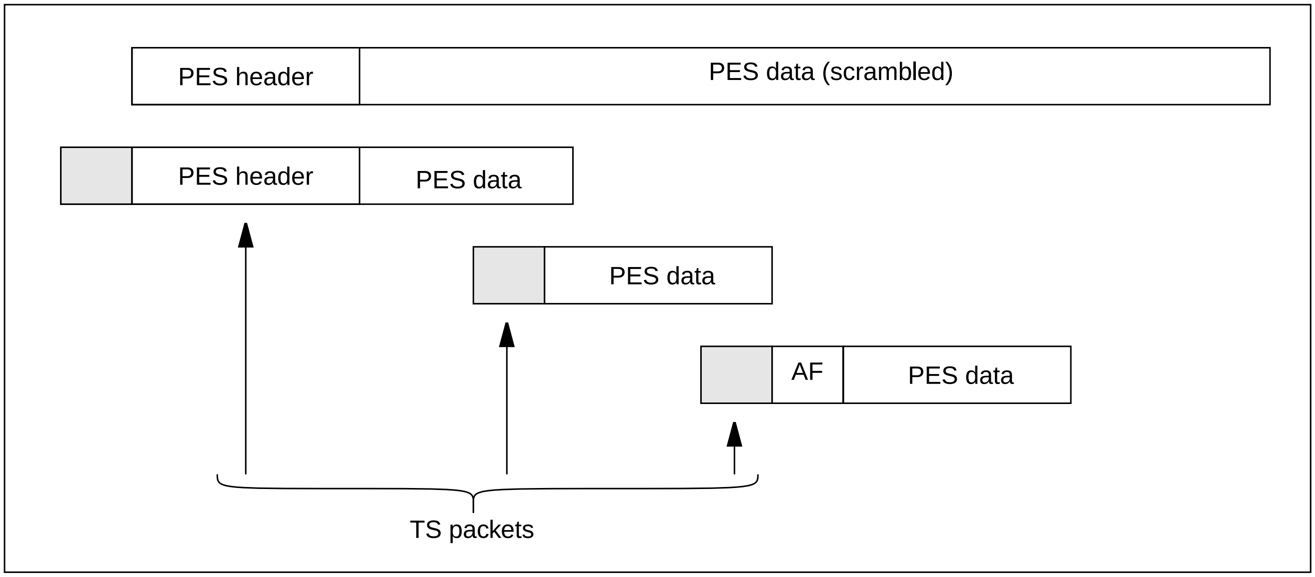


Figure 16.9 PES level scrambling (DVB recommendations)

### 16.2.3 SDAV interface

This block is used to send a single stream out to a high speed serial digital bus or playback a stream from that bus. There are two bus formats supported (SDAV and IEEE1394). The bus formats are described below. The time stamp, which is used to maintain proper packet placement, is the value of the LSBs of a continuously running 27MHz clock. This time stamp is added for SDAV bus and optionally for IEEE 1394 bus in tape-out mode, but it is simply discarded when received from these busses in tape-in mode. The 12 reserved bits are added from the **EXTRA_BITS_REG**.

### SDAV format

The method of transmission on the SDAV is using a 49.152MHz bit rate in a non-return-to-zero (NRZ) encoding method. The signals that are used are the following:

- **strobe_tx**, **data_tx**, direction.

The signals **strobe_tx** and **data_tx** are reversed for transmit and receive. That means the signal **strobe_tx** acts as the data signal in receive mode and strobe in transmit mode. And the signal **data_tx** acts as the strobe signal in receive mode and data in transmit mode. The signal direction controls the mode. When direction is high, the mode is transmit and the other two signals are outputs (see Figure 16.10).

The functionality of the clock pin **Sdav_Clk** is changed according to the register bits **SDAV_1394**, **EXT_CLK** in the **SDAV_CONF_REG** and **DVD_MODE** and **PCM_MODE** in the **MODE_REG**.

### IEEE 1394 format

For IEEE 1394 bus operation, the STi5500 must be used in conjunction with an IEEE 1394 IC that will interface to the physical bus. This section describes the interface between the STi5500 IC and the IEEE 1394 IC. The signals that are used are the following:

- **clock**,
- **data**,
- **data_valid** (packet clock).

All three signals are outputs for tape-out mode and inputs for tape-in mode. The clock is continuous and the **data_valid** is active for the entire packet without gaps between the bytes (see Figure 16.11).

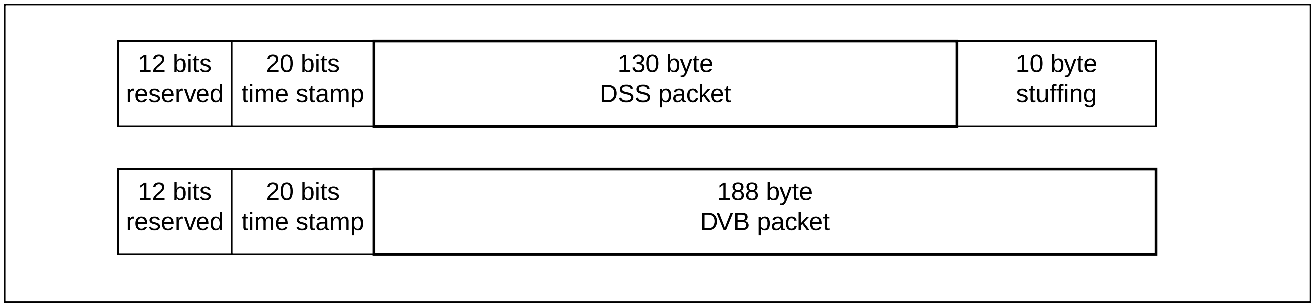


Figure 16.10 Format for DSS and DVB in SDAV

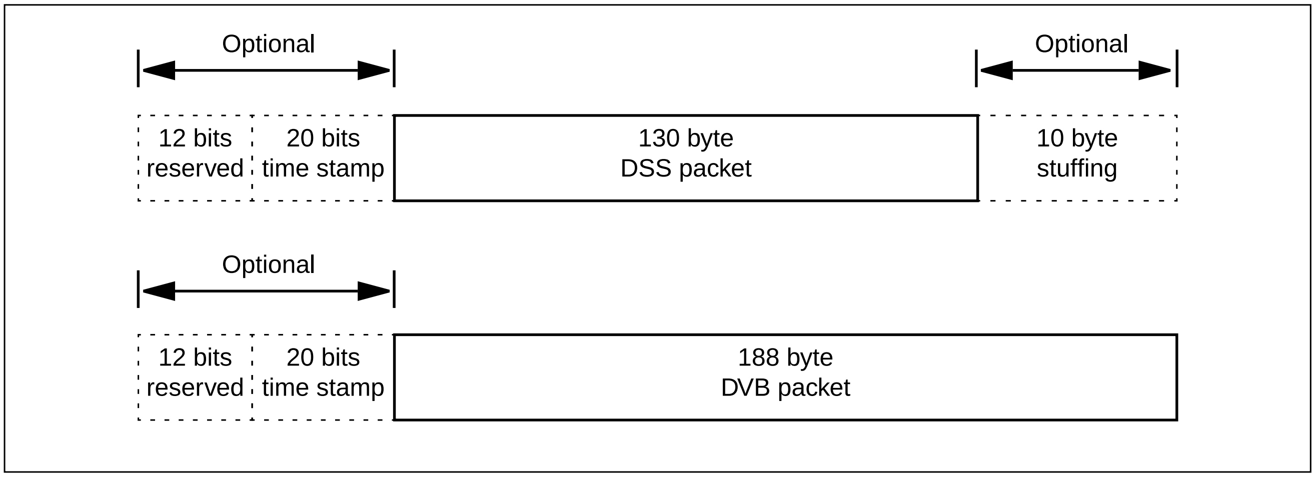


Figure 16.11 SDAV interface processing

### 16.2.4 Packet formatting for SDAV and IEEE 1394

If an external IEEE 1394 chip is connected, the format on the SDAV bus is modified (see Table 16.7). The NRZ encoding and decoding do not take place in IEEE 1394 mode.

| Bus | Mode | Incom-plete DMA | Header_ Enable | Stuffing _ Enable | Header Bytes | Tp Bytes | Padding Bytes | Stuffing Bytes | TOTAL |
|---|---|---|---|---|---|---|---|---|---|
| SDAV | DVB | 0 | X | X | 4 | 188 | | | 192 |
| SDAV | DVB | 1 | X | X | 4 | X | 188-X | | 192 |
| SDAV | DSS | 0 | X | X | 4 | 130 | | 10 | 144 |
| SDAV | DSS | 1 | X | X | 4 | Y | 130-Y | 10 | 144 |
| 1394 | DVB | 0 | 0 | X | | 188 | | | 188 |
| 1394 | DVB | 0 | 1 | X | 4 | 188 | | | 192 |
| 1394 | DVB | 1 | 0 | X | | X | 188-X | | 188 |
| 1394 | DVB | 1 | 1 | X | 4 | X | 188-X | | 192 |
| 1394 | DSS | 0 | 0 | 0 | | 130 | | | 130 |
| 1394 | DSS | 0 | 0 | 1 | | 130 | | 10 | 140 |
| 1394 | DSS | 0 | 1 | 0 | 4 | 130 | | | 134 |
| 1394 | DSS | 0 | 1 | 1 | 4 | 130 | | 10 | 144 |
| 1394 | DSS | 1 | 0 | 0 | | Y | 130-Y | | 130 |
| 1394 | DSS | 1 | 0 | 1 | | Y | 130-Y | 10 | 140 |
| 1394 | DSS | 1 | 1 | 0 | 4 | Y | 130-Y | | 134 |
| 1394 | DSS | 1 | 1 | 1 | 4 | Y | 130-Y | 10 | 144 |

Table 16.6

**HEADER** if SDAV or (1394 and **header_enable**) and not (PCM or DVD).

**PADDING** if incomplete DMA and not (PCM or DVD).

**STUFFING** if DSS and (SDAV or (1394 and **header_enable**)).

| Signals | 1394 | | SDAV | |
|---|---|---|---|---|
| | Direction | Description | Direction | Description |
| DATA | I/O | Data | I/O | DATA_TX/STROBE_RX |
| DATA_CLOCK | I/O | Clock | I/O | STROBE_TX/DATA_RX |
| PACKET_CLOCK/DIRECTION | I/O | PACKET_CLOCK(1) | O | Direction |

Table 16.7 External interface

Note: **PACKET_CLOCK** (valid data) is active during the TP packet. IN or OUT for IEEE 1394 mode.

| | Pin | | IEEE 1394 | | SDAV | |
|---|---|---|---|---|---|---|
| | Dir | Name | Dir | Description | Dir | Description |
| In | I/O | **DATA** | I | DATA_IN (No NRZ Decoding) | I | STROBE_RX (49.1MHz) (NRZ Decoding) (Only Header + Payload) |
| | I/O | **CLK** | I | Continuous (Up to 60MHz) | I | DATA_RX (NRZ Decoding) |
| | I/O | **PACK_CLK_DIR** | I | Data Valid (Packet Clock) | O | Direction (Tape In) |
| Out | I/O | **DATA** | O | DATA_OUT (No NRZ Encoding) | O | DATA_TX (NRZ Encoding) |
| | I/O | **CLK** | O | Continuous (60MHz) | O | STROBE_TX (49.1MHz) (NRZ Encoding) (Only Header + Payload) |
| | I/O | **PACK_CLK_DIR** | O | Data Valid (Packet Clock) | O | Direction (Tape Out) |

Table 16.8 Use of the three I/O pins

### 16.2.5 Data Path

The maximum input rate is about 7.5Mbytes/s (7/8 x 68Mbits/s). The data are sent to the SDAV interface either scrambled or not. It should be noticed that the descrambler works up to 60Mbits/s. Null packets are not transmitted to the digital bus. Packets concerning other programs and any unwanted information are also discarded. In such cases, packets may be generated by software and transmitted by DMA transfer, for example. Those packets must be isochronous with the packets extracted from the original multiplex. Some tables (PAT, PMT) may be modified by S/W to create new program guides for use in playback mode

### 16.2.6 Tape-In

In Tape-in mode the SDAV interface serves as a data source similar to and instead of the FEC input. The data will be received from the interface and sent to the NRSS block and into the acquisition RAM. The data can be sent directly or resynchronized to **Sys_Clk**. Any header or stuffing if enabled are stripped from the packet and discarded with the exception of the 12 reserved bits in the header. These reserved bits are latched in the **EXTRA_BITS_REG**. If these bits differ from the contents of the register prior to the latching, an interrupt is generated.

The **SDAV_OVERFLOW** interrupt is generated to indicate to the CPU that some extra bits are available. Both fields (**SDAV_OVERFLOW** and **SDAV_UNDERFLOW**) are set in the **LINK_STAT_FIFO**. This Interrupt is maskable by **EXTRA_BITS_REG.EBM** and is generated only when the incoming extra_bits change value.

| EXTRA_BITS_REG | Comment |
|---|---|
| 12:1 | EXTRA_BITS_INPUT |
| 12:5 | Playback Rate Control |
| 4:1 | Copy Guard Information |
| 0 | EXTRA_BITS_IRQ_MASK |

Table 16.9 Format of register **EXTRA_BITS_REG**

| | SDAV Input | SDAV Output |
|---|---|---|
| **SDAV_OVERFLOW** | EXTRA_BITS_IRQ | **SDAV_OVERFLOW** |
| **SDAV_UNDERFLOW** | when both = 1 | **SDAV_UNDERFLOW** |

Table 16.10

### 16.2.7 Tape-out

In tape-out mode the SDAV block receives data from either the acquisition RAM or from the descrambler. The Transport Stream Demultiplexor system clock (**Sys_Clk**) will be used as CLK_IN (60 MHz). The output clock will be the interface clock (49.1 for SDAV, up to 60 MHz for IEEE 1394). The input data stream speed will vary with the speed of the incoming FEC data or it will be at whatever speed the DMA engine can provide.

The data are latched (at the CLK_IN frequency) into a single port RAM to guarantee the output of one complete packet at the corresponding clock frequency. It means that the SDAV block will receive about one byte every 8 clock cycles (CLK_IN). As soon as there is enough information in the RAM (not to run out of data before the end of the packet), the SDAV Interface generates the header information and then converts the data to the SDAV bus serial format.

### 16.2.8 CPU generated packets

The CPU can create custom program guide packets for insertion into the bitstream and DMA the packets to the SDAV block.

The packets will be inserted in the out-going stream where possible. When a DMA is set up to send data to the SDAV block, it is always set up to transfer 32-bit words. Before enabling the DMA, the CPU must set the value of the **FIRST_BYTE_POSIT** and **LAST_BYTE_POSIT** fields in the **SDAV_DMA_EN_REG** register.

The **FIRST_BYTE_POSIT** should be loaded with the two least significant bits of the address of the first byte to be transferred. The **LAST_BYTE_POSIT** should be loaded with the two least significant bits of the address of the last byte to be transferred. If the data that is being sent starts at address 0x40001000 and ends at address 400010FF, the value of **FIRST_BYTE_POSIT** is 00 and the value of **LAST_BYTE_POSIT** is 0x11. If the data to be sent starts on an odd boundary such as 0x40001001, the DMA should start with the address 0x40001000 but the **FIRST_BYTE_POSIT** should be loaded with 01. Similarly if the data to be sent ends on an odd boundary such as 0x400010FE, the DMA should transfer the entire 32 bit word starting at address 0x400010FC but the **LAST_BYTE_POSIT** should be loaded with 10.

### First and last bytes for DMA transfer to the SDAV interface block

See Figure 16.12.

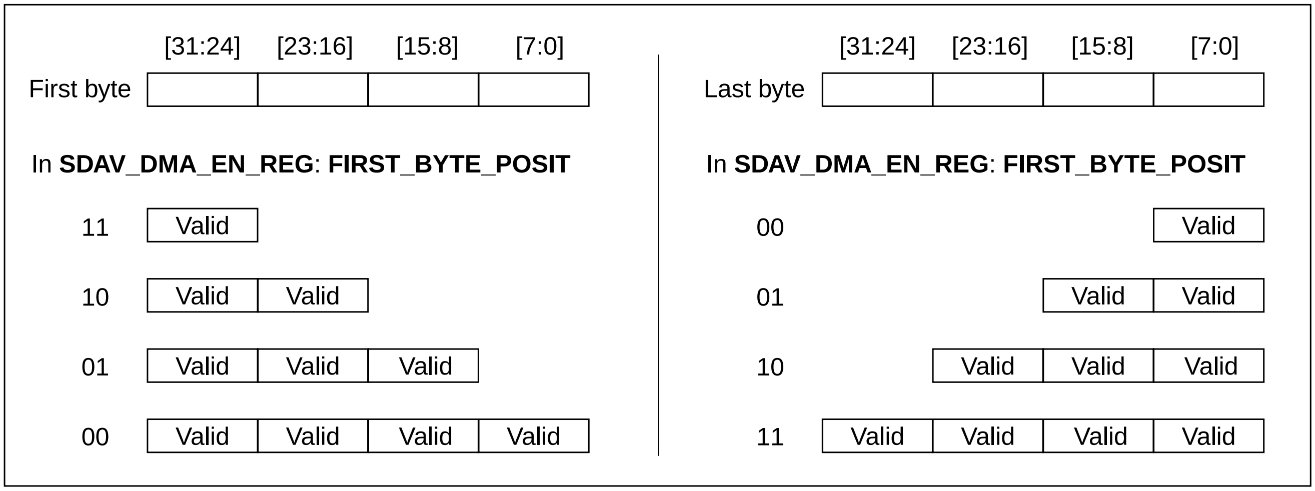


Figure 16.12 TP and PES headers

## 16.3 FRAM

### 16.3.1 RAM (480x32 bit)

The FRAM is a dual port RAM of 480 x 32 bits that holds the complete information of the Transport Stream Demultiplexor. This includes the filter information, the stream configurations, the keys for the descrambler and the IRQ words, as shown in Figure 16.13.

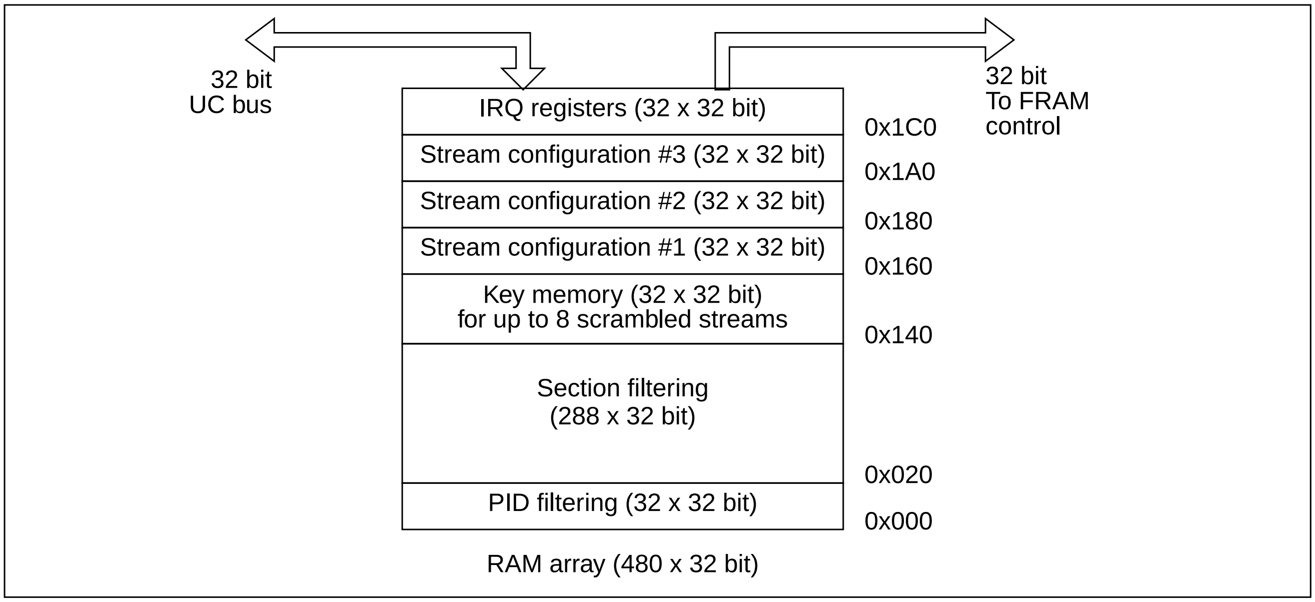

32 bit
UC bus
32 bit
To FRAM
control
IRQ registers (32 x 32 bit)
Stream configuration #3 (32 x 32 bit)
Stream configuration #2 (32 x 32 bit)
Stream configuration #1 (32 x 32 bit)
Key memory (32 x 32 bit)
for up to 8 scrambled streams
Section filtering
(288 x 32 bit)
PID filtering (32 x 32 bit)
0x1C0
0x1A0
0x180
0x160
0x140
0x020
0x000
RAM array (480 x 32 bit)

Figure 16.13 FRAM organization

One port is used by the micro to initialize the RAM. The other port is used by the processors, the filter and the descrambler. After the PID is filtered, the stream number is used to generate the address for the stream initialization. This configuration determines further processing of the stream, such as section, error code and filtering.

In order to save power, the FRAM is only enabled when an access from one of the processors occurs. This happens whenever there is PID filtering, AF filtering, section filtering, the loading of the stream configuration, the saving of the stream configuration at the end of the packet (section length if section is over 2 TP, CC...) and the IRQ status word read and write operations.

### 16.3.2 Filtering

The filtering is done by precalculating the result. The byte to be filtered is used to generate the address in the FRAM. A '1' at an address means a match. For an 8-bit value, this would give 256 bits in the RAM (see Figure 16.14).

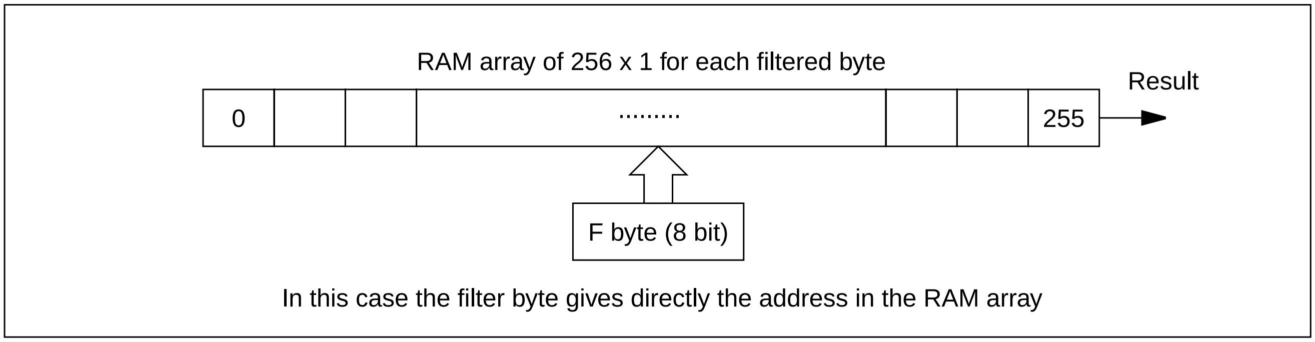

RAM array of 256 x 1 for each filtered byte
Result
0
.........
255
F byte (8 bit)
In this case the filter byte gives directly the address in the RAM array

Figure 16.14 Basic principle of filter mechanism

To reduce the RAM size, the filtering is done in four steps per incoming byte. In each step, two bits of the incoming byte plus a 2-bit pointer generate the address in the RAM. This needs 16 bits of RAM for one filter byte.

Each bit of the byte to be filtered can be masked individually as described below (see Table 16.11).

As the RAM is 32 bits wide, we can filter on 32 targets in parallel. However it is also possible to filter on less then 32 targets. In this case the multiplexor at the output of the FRAM allows selection of only a part of the data. With each new byte to be filtered, the address of this multiplexor changes (by adding the filter number) and therefore selects a different part of the output data.

Therefore, the lowest bit of the filter match register holds the result of the filter process. For example, if only one target is filtered, the LSB of the register holds the result.

### 16.3.3 Error Procedures

On the input of the STi5500, the packet stream can be erroneous. Several error mechanisms are applied depending on error that occurs. Table 16.11 is a list of the possible errors and the applied mechanisms (see Table 16.12).

| Filter Mask | Value in FRAM | Filter Mask | Value in FRAM |
|---|---|---|---|
| 01101100<br>11111111 | 0100 0010 0001 1000 | ... | |
| 0110110<br>11111110 | 0100 0010 0001 1100 | 01101100<br>00000010 | 1111 1111 1111 1100 |
| 01101100<br>11111101 | 0100 0010 0001 1010 | 01101100<br>00000001 | 1111 1111 1111 1010 |
| ... | | 01101100<br>00000000 | 1111 1111 1111 1111<br>All Input Bytes Are Valid |

Table 16.11

See also not-equal filtering mode.

| Error | Mode | Description |
|---|---|---|
| FEC | DVB<br>DSS | This signal is delivered by the Link IC and signals a packet error. In this case the transport packet is not processed, it is not written into the AR. |
| SYNC_BYTE | DVB | If the sync_byte in the packet header is not correct (≠ 0x47), the TP is rejected. |
| TRANSPORT_ERROR _INDICATOR | DVB | This bit belongs to the TP header; if it is set the TP is rejected. This is done with the filter process. |
| CC | DVB<br>DSS | If the received CC does not match the expected one, different mechanisms are applied according to the stream type. |

Table 16.12

The CC error code insertion can also be switched on or off for each stream individually by the stream configuration.

| Event | Action |
|---|---|
| Suspend = '0' | No CC error generated |
| CC Error and ERROR_PATT = '1' and not (only AF) and PES | Error code is generated: B4 00 00 01 B4 |
| CC Error for a Section Stream | If section is active, the stream is disabled and IRQ is generated (see below) |

Table 16.13

Table 16.14 summarizes the CC processing for DVB.

| AF | Payload | Event | Transport Stream Demultiplexor processing |
|---|---|---|---|
| 0 | 0 | CC is incremented or not | Skipped TP - dummy transfer |
| 1 | 0 | CCn = CCn-1 | End of CC processing |
| | | CCn ≠ CCn-1 | $CC_{stored}$ is modified by the Transport Stream Demultiplexor.[1] |
| 0 | 1 | CC is incremented | End of CC processing |
| | | CCn = CCn-1 | Duplicated TP: dummy transfer |
| | | CC is not incremented and CCn ≠ CCn-1 | • Section + suspend = 1: stream disabled.<br>• Section + suspend = 0: nothing<br>• PES + suspend = 1: insert error code<br>• PES + suspend = 0: nothing |
| 1 | 1 | Same processing as previous combination (2) | |

Table 16.14 CC processing for DVB

[1] This will create a CC error for the next packet on this PID.

[2] If the discontinuity indicator is set (in the AF) and if an error code was inserted, this code must be removed.

## 16.4 Not-equal filtering

A special mode allows filtering on a not-equal condition (defined in **Stream_conf_1**). The byte to which this is applied is programmable. It can also be done in parallel to up to 8 different targets. When not-equal filtering, one additional byte has to be filtered after the not-equal byte.

After the section length, the **SEC_F_INV_REG** is loaded. It cannot be loaded with 111. The function is activated with the **SEC_F_INV** signal (see Figure 16.15).

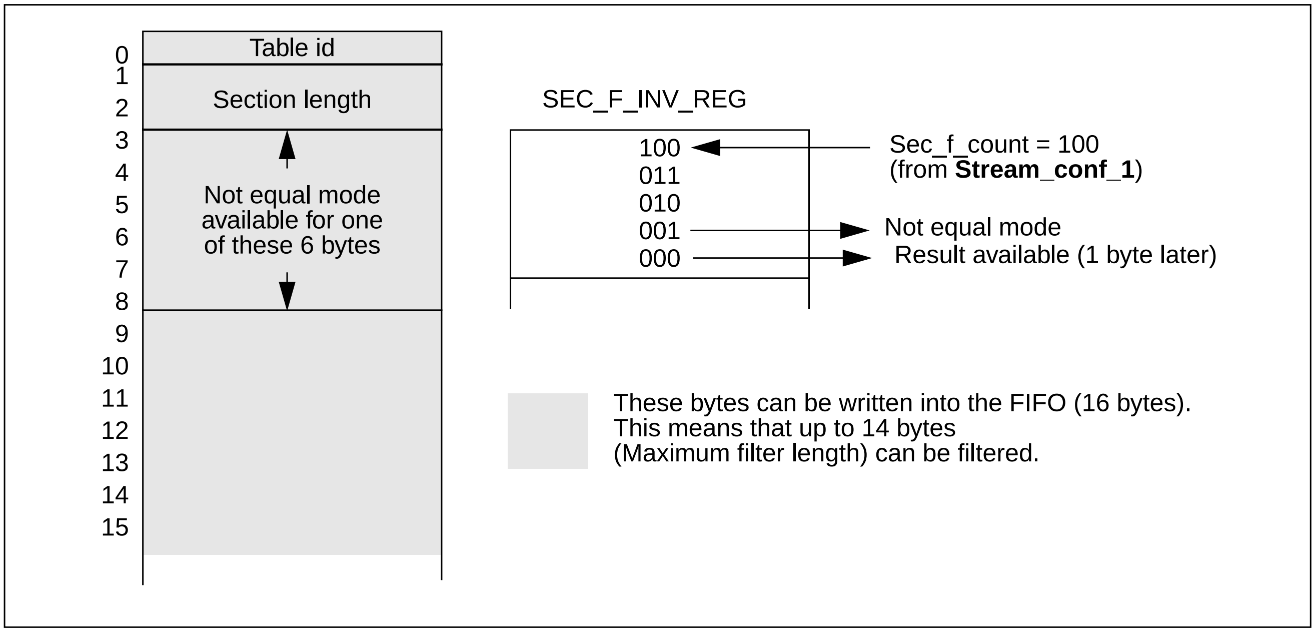

0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
Table id
Section length
Not equal mode available for one of these 6 bytes
SEC_F_INV_REG
100
011
010
001
000
Sec_f_count = 100
(from Stream_conf_1)
Not equal mode
Result available (1 byte later)
These bytes can be written into the FIFO (16 bytes).
This means that up to 14 bytes
(Maximum filter length) can be filtered.

Figure 16.15 Section filtering example

One byte of each target can be checked to be different from a specific value. At the beginning, all sections can be received. This is done by masking the specified byte (the FRAM is initialized with 0x00 in not_equal mode). After each section has arrived, the filter for the specified byte can be written into the FRAM (see Table 16.15).

| Filter Mask | Value in FRAM | Equal mode | Not-equal mode |
|---|---|---|---|
| 01101100<br>11111111 | 0100 0010 0001 1000 | 01101100 valid | All Valid Except<br>01101100 |
| 01101100<br>01101100 | 0100 0010 0001 1100 | 01101100 and<br>01101101 valid | All valid except<br>01101100 and<br>01101101 |
| 01101100<br>10011111 | 1100 1010 0001 1010 | 01101100 and<br>00101100 and<br>01001100 and<br>00001100 valid | All valid except<br>01101100 and<br>00101100 and<br>01001100 and<br>00001100 valid |
| ... | ... | ... | ... |
| 01101100<br>00000000 | 1111 1111 1111 1111 | All bytes valid | No byte valid |
| 01101100<br>???????? | 0000 0000 0000 0000<br>At least 1 nibble = 0000 | No byte valid | All bytes valid |

Table 16.15

Note: 0 means the bit is masked
1 means the bit is not masked

## 16.5 DMA

MPEG audio, video and system data can be transferred to any location in the ST20 address space (internal SRAM, external SDRAM or DRAM, MPEG decoders) via a DMA controller.

The DMA transfers the data from the Transport Stream Demultiplexor to a destination address which can be set individually for each stream.

| Register | Bit field | Bits | Circular buffer |
|---|---|---|---|
| **Stream_conf_2** | **BUFFER_SIZE** | 30:28 | **CIRCULAR_BUFFER_SIZE** |
| | **STOP_ADDR** | 27:18 | **DMA_STOP_ADDR (14:5)** |
| | **START_ADDR** | 17:11 | **DMA_START_ADDR (18:8)** |
| | **DMA_BANK_ADDR** | 10:7 | **DMA_ADDRESS (31:28)** |
| | **DMA_HIGH_ADDR** | 6:0 | **DMA_ADDRESS (21:15)** |
| **Stream_conf_3** | **DMA_LOW_ADDR** | 26:12 | **DMA_ADDRESS (14:0)** |

Table 16.16 DMA configuration

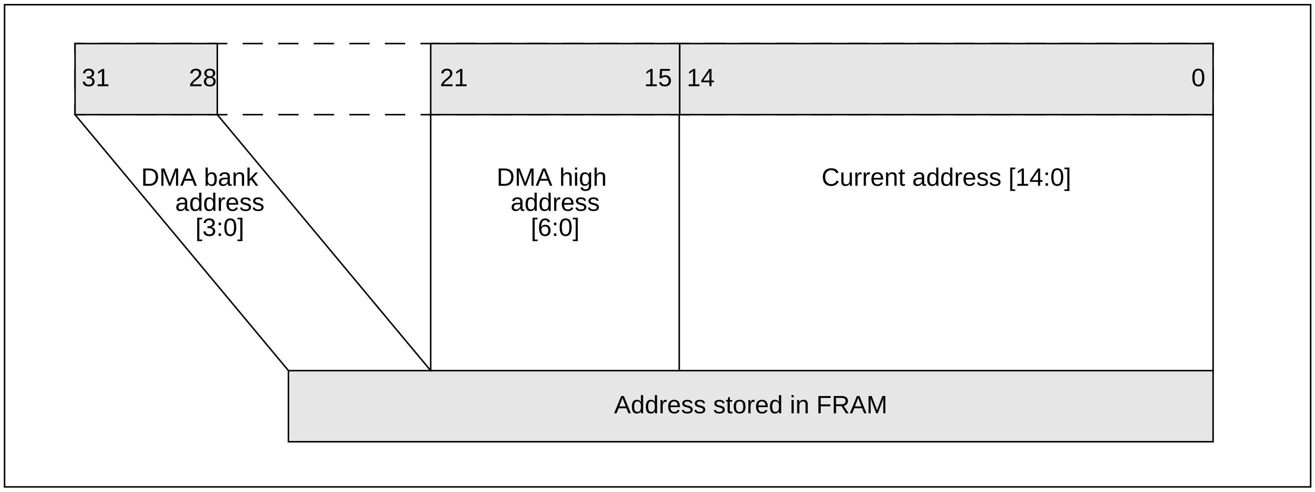


Figure 16.16 DMA address

### 16.5.1 DMA Description

At the beginning of the packet, the DMA is initialized by **Stream_conf_2** and **Stream_conf_3**. There are two basic modes of the DMA - incremental and non-incremental. For DVD, incremental mode has two sub-modes - circular and linear.

At the end of the packet, the current address of the DMA can be stored back to the FRAM. It will be reloaded by the Transport Stream Demultiplexor when the next packet on this stream arrives. The address write-back is not done in DVD mode.

If the increment bit of **Stream_conf_2** is set to '1', and the **DISABLE_FINAL_BURST** bit of **MODE_REG** is set to '0', the last transfer will be a burst transfer where the not used bytes are filled with dummy data. If the increment is set to '0', or the **DISABLE_FINAL_BURST** is set to '1', the exact number of bytes is transferred (MPEG decoders).

However, the address counter will keep the value of the last valid byte that is transferred. Two buffers are used in the DMA (two times 4x32 bits). While one buffer is transferred the second one is filled. This allows data to be read from the Transport Stream Demultiplexor even if the DMA has to wait for the ST20 bus.

The address pointer which will normally increment every time something is transferred, has to be limited in order not to destroy information from other buffers or program code. To this end circular buffers are implemented.

For all buffers:

| | |
|---|---|
| **END_ADDR** | = **START_ADDR** + SIZE - 1 |
| **CURRENT_ADDRESS** | = first byte to be written<br>= the value in **Stream_conf_3** at the end of packet |
| **STOP_ADDRESS** | = last byte read |

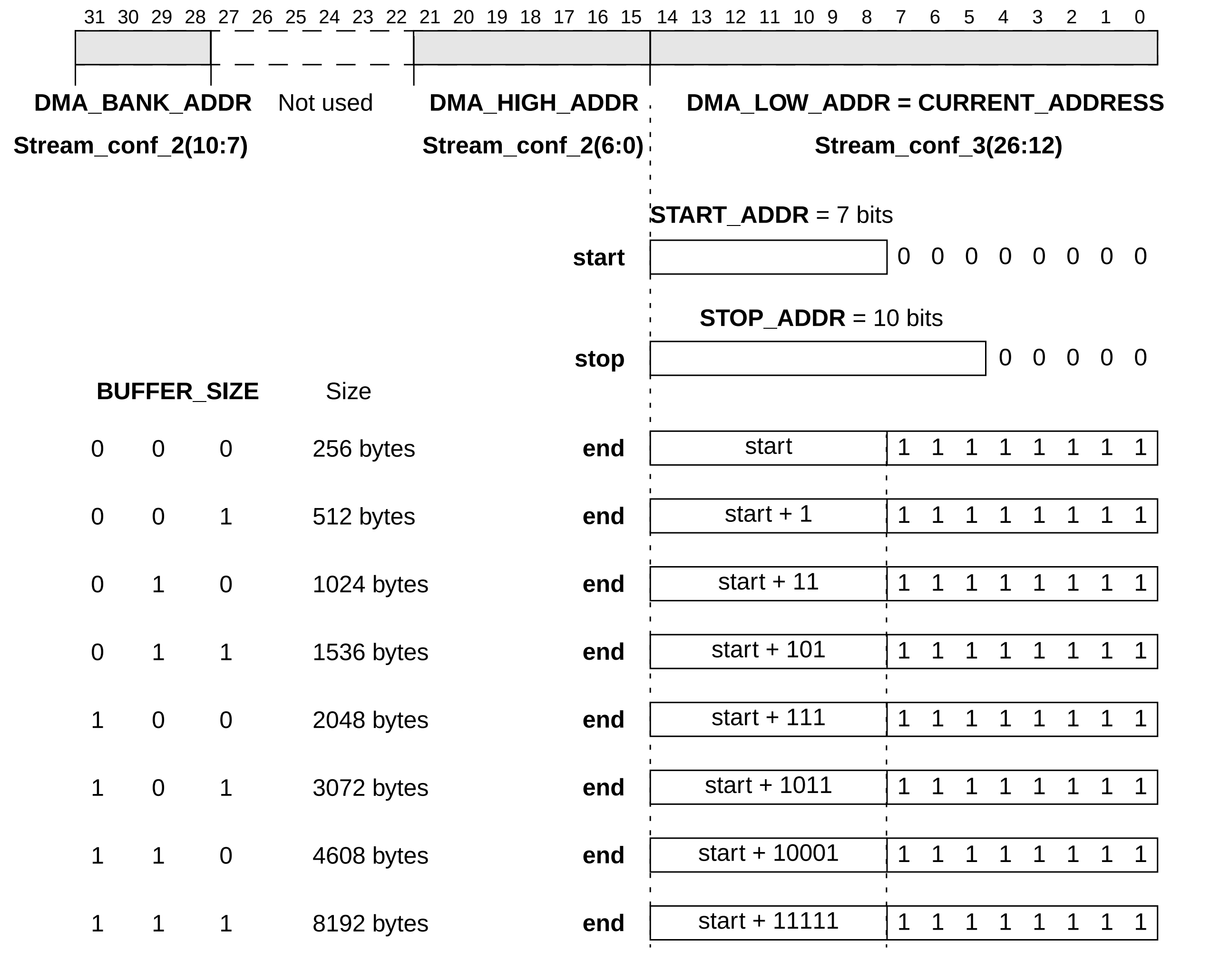
### The circular buffer (incremental mode)

To stop properly the transfer when the circular buffer is full, the CPU writes the **STOP_ADDR** in the Transport Stream Demultiplexor.

- In this case when the transfer pointer (**START_ADDR**) reaches this value, the DMA can be aborted. This will prevent overwriting of data which has not been processed.
- The CPU updates the **STOP_ADDR** each time it has finished processing data.
- A **START_ADDR(2:0)** and a **STOP_ADDR(9:0)** are specified. The meaning of these bits depends on the **BUFFER_SIZE(2:0)**, as shown in Figure 16.17 and Figure 16.18).
- When the transfer address reaches the top of the circular buffer, it is reset to the bottom of the circular buffer and the transfer continues.
- If the transfer is aborted (if **STOP_ADDR** is reached), the DMA engine generates a DMA overflow. When this occurs, the rest of the packet is discarded. The DMA buffer is flushed to its destination and the **Stream_conf_3** is saved back to the FRAM. The stream is automatically disabled. The **DMA_overflow** bit is set with the **STREAM_NUMBER** in the status word, which is written into the **LINK_STAT_FIFO**.

| **BUFFER_SIZE (2:0)** | **Buffer size (bytes)** | **Stop precision (bytes)** |
|---|---|---|
| 000 | 256 | 32 |
| 001 | 512 | 32 |
| 010 | 1024 | 32 |
| 011 | 1536 | 32 |
| 100 | 2048 | 32 |
| 101 | 3072 | 32 |
| 110 | 4608 | 32 |
| 111 | 8192 | 32 |

Table 16.17 DMA buffer sizes



| BUFFER_SIZE | | | Size | | | |
|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 256 bytes | end | start | 1 1 1 1 1 1 1 1 |
| 0 | 0 | 1 | 512 bytes | end | start + 1 | 1 1 1 1 1 1 1 1 |
| 0 | 1 | 0 | 1024 bytes | end | start + 11 | 1 1 1 1 1 1 1 1 |
| 0 | 1 | 1 | 1536 bytes | end | start + 101 | 1 1 1 1 1 1 1 1 |
| 1 | 0 | 0 | 2048 bytes | end | start + 111 | 1 1 1 1 1 1 1 1 |
| 1 | 0 | 1 | 3072 bytes | end | start + 1011 | 1 1 1 1 1 1 1 1 |
| 1 | 1 | 0 | 4608 bytes | end | start + 10001 | 1 1 1 1 1 1 1 1 |
| 1 | 1 | 1 | 8192 bytes | end | start + 11111 | 1 1 1 1 1 1 1 1 |

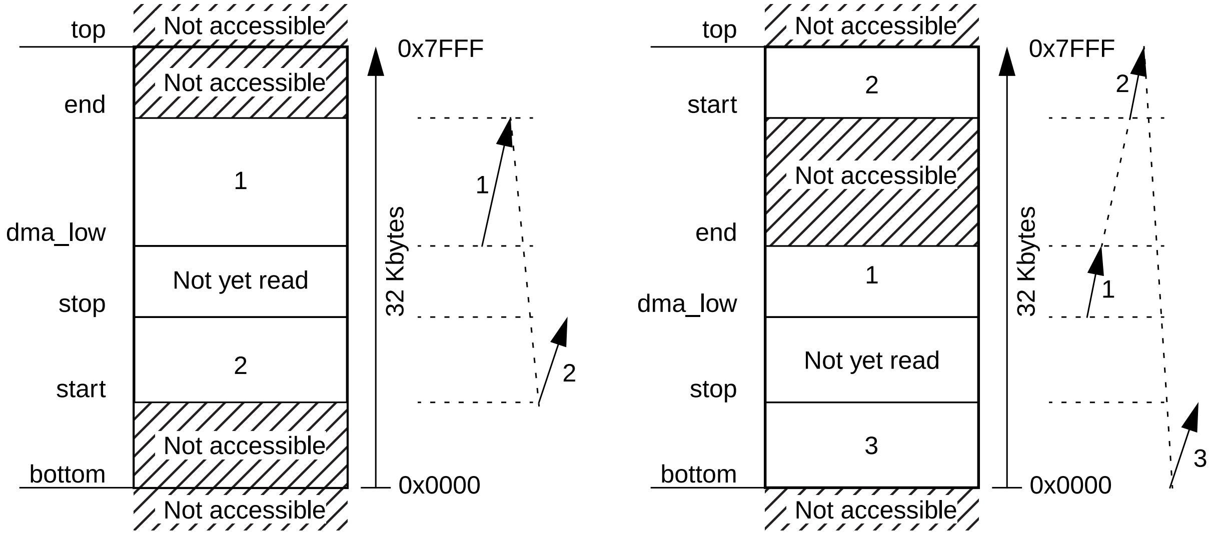
Figure 16.17 Buffer size definition

### The DVD buffer (linear mode)

In DVD mode, the DMA can be incremental or non-incremental, but if it is incremental, the circular buffer is not used. The address, starting from **DMA_LOW_ADDR**, is incremented after each access, independently from the start and the stop addresses if the increment bit is set to '1'. If the increment bit is set to '0', all the data is written to the address specified by the **DMA_HIGH_ADDR** and the **DMA_LOW_ADDR**.

### The non-incremental buffer

This mode is used when addressing the CD FIFOs. The **CURRENT_ADDRESS(1:0)** is incremented by 1 after each access.



1, 2 and 3 are locations that are writable

Figure 16.18 Circular buffer diagram

| Buffer | Circular buffer | DVD buffer | Non-incremental Buffer |
|---|---|---|---|
| **BUFFER_SIZE** | See Table 16.16 | See Table 16.16 | Not used |
| **STOP_ADDR** | | Not used | |
| **START_ADDR** | | | |
| **DMA_BANK_ADDR** | Used to initialize the DMA. | | |
| **DMA_HIGH_ADDR** | | | |
| **DMA_LOW_ADDR** | | | |

Table 16.18  DMA address

## 16.6 Clock recovery

This block helps the control unit to perform the clock recovery and clock synchronization processes. It uses local counters clocked with the video clock (27MHz).

The value of these counters can be latched in four different registers that can be read by the control unit.

**PCR_EXT_REG(8:0)** and **PCR_REG(31:0)** are updated with the local time counters when a new packet occurs. If an AF with PCR is found (if PCR flag is present in AF byte #0), a sample of the 27Mhz clock is latched and the first 8 bytes of the incoming Adaptation Field are stored in **AF_REG[1:0]**. It allows the controller unit to synchronize the decoder clock reference (27MHz). A new PCR cannot be latched if the current PCR (**AF_REG1**) has not been read by the CPU.

When a rising edge is detected on the latch from the video decoder or audio decoder, the appropriate Transport Stream Demultiplexor register, **V_PTS_REG** or **A_PTS_REG** respectively, is updated with the local time **Counter(31:0)**. The latches are reset by the clock recovery entity after the register has been read by the control unit.

A counter(19:0) is also required to generate the time stamp used for the SDAV header. The time stamp value is updated at the beginning of each packet.

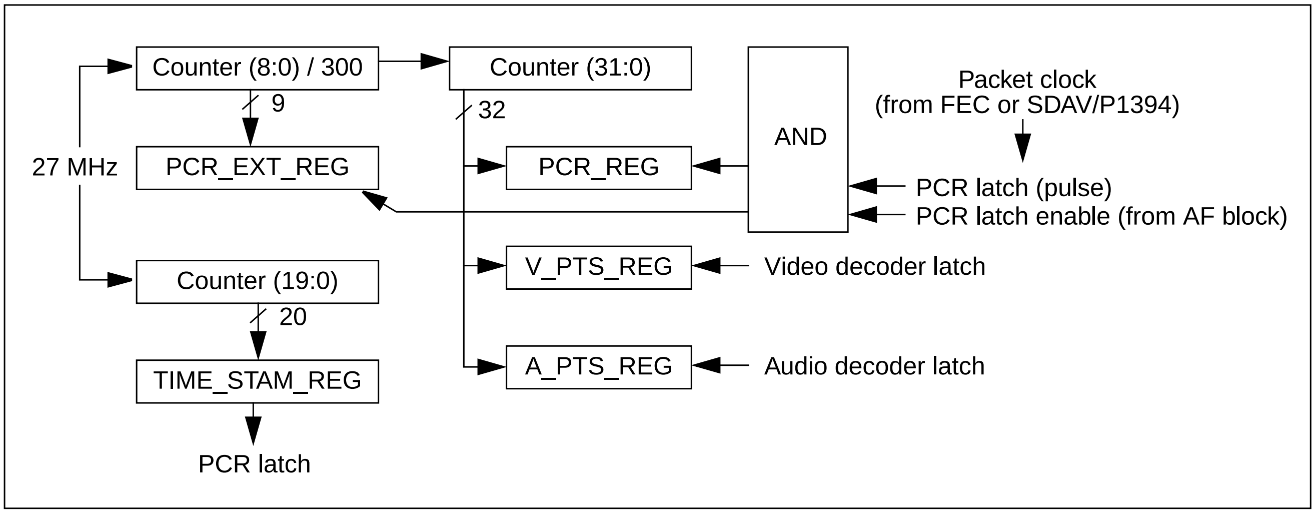


Figure 16.19 Clock recovery

## 16.7 Interrupts

Table 16.19 shows the interrupt sources specified for each stream.

| Number | Interrupt | Maskable | Stream disabled |
|---|---|---|---|
| 1 | AR Overflow | no | no |
| 2 | DMA Overflow | no | yes |
| 3 | Bad Section | no | yes |
| 4 | End of Section Filtering | yes | no |
| 5 | End of Section Transfer | yes | no |
| 6 | Incomplete Filtering | yes | no |
| 7 | AF | yes | no |
| 8 | SDAV Underflow | no | no |
| 9 | SDAV Overflow | no | no |

Table 16.19 Interrupt sources

### 16.7.1 IRQ FIFO

#### Global Mechanism

Each interrupt source has a corresponding status bit in the status word **LINK_STAT_FIFO**. If the interrupt is enabled (mask=1) and the corresponding status bit is set, an interrupt event is generated.

Successive interrupts generate successive status words **LINK_STAT_FIFO** that are stored, in chronological order, in a 32-word FIFO. This is useful if multiple interrupts occur within a packet (multiple sections per packet). The FIFO is implemented as a circular buffer in the FRAM.

#### LINK_STAT_FIFO

In the status word, the **STREAM_NUMBER** field and interrupt status bits of the status word are always applicable. The **TARGET_MATCH** and **OTHER_MATCH** bits are only valid on **EOF** and **INCOMPLETE_FILTER** interrupts. In addition, the **FILTER_OFFSET** field is only valid on **INCOMPLETE_FILTER** interrupts.

#### LINK_STAT_REG

The **FIFO_NOT_EMPTY** field of the **LINK_STAT_REG** register indicates if interrupts are pending. Each time a status word is written to (or read from) the FIFO, a write (or read) pointer is incremented in **LINK_STAT_REG**. The FIFO is read by the software through the **LINK_STAT_FIFO** register. When a word is read, it is removed from the FIFO.

The software should check the **LINK_STAT_REG** for FIFO emptiness or overflow before reading the **LINK_STAT_FIFO**. The software should not read the FIFO if it is empty.

When at least one status word is present in the **LINK_STAT_FIFO**, the interrupt line to the CPU is set active, and kept active as long as the FIFO is not empty. The interrupt line becomes inactive when the last word is read from the FIFO.

If the FIFO is full and new interrupt events occur, no further status word is written to the FIFO and the **FIFO_OVERFLOW** bit of the **LINK_STAT_REG** register is set. This bit is reset when the software reads a word out of the FIFO.

### 16.7.2 AR Overflow

#### IRQ Generation

An AR overflow occurs if at least one of the following condition is true:

- a new packet has started being stored in the AR and its processing has not started as **AR_timeout** bytes have been stored.
- the write pointer of the AR reaches the read pointer.

This can happen if the Transport Stream Demultiplexor hangs or if the ST20 bus traffic prevents data to be output fast enough.

#### IRQ Processing

The stream is not disabled.

When such an AR overflow occurs, the current packet processing is aborted. All packet data which have not already been passed to the DMA write buffer are discarded. The DMA write buffer is flushed to its destination.

An internal reset signal is generated to put the whole Transport Stream Demultiplexor into a state where it expects a new packet to arrive:

- if a start of packet is present in the AR, data input is not stopped and overwrites the discarded data.
- if a start of packet is already present in the AR, its processing starts immediately.

The **Stream_conf_3** of the aborted packet is not saved back to FRAM. Therefore appropriate error processing will be performed due to CC discontinuity on the next packet of the same PID.

The **AR_OVERFLOW** bit is set along with the **STREAM_NUMBER** in the status word and the status word is written into the **LINK_STAT_FIFO**.

### 16.7.3 Storage Buffer Overflow

#### IRQ Generation

A DMA overflow interrupt occurs when the write pointer of the DMA transfer reaches the stop value stored in **Stream_conf_2**.

#### IRQ Processing

The stream is disabled.

The rest of the packet is discarded. The DMA buffer is flushed to its destination.

**Stream_conf_3** is saved back to FRAM.

The DMA_overflow bit is set along with the **STREAM_NUMBER** in the status word and the status word is written into the **LINK_STAT_FIFO**.

### 16.7.4 Bad Section

#### IRQ Generation

When saving a section to memory, the Transport Stream Demultiplexor counts the section length and knows therefore the end of the section.

The following conditions will generate a bad section interrupt:

- The PUS of the current packet is set and the pointer_field at the beginning of the packet does not correspond to the number of remaining bytes in the currently transferred section.
- The CC of the current packet has the wrong value on a section packet when a section is being processed.
- The PUS is not set and the bytes following the current section are not stuffing bytes (FF).

#### IRQ Processing

The stream is disabled.

In such condition, the bad_sec bit is set along with the **STREAM_NUMBER** in the status word and the status word is written into the **LINK_STAT_FIFO**.

(Note that the CC is checked on PES streams if the error_patt bit of the **Stream_conf_1** is set. But no IRQ is generated there.)

### 16.7.5 End of Section Filtering

#### IRQ Generation

- DSS/DVB mode

  An eof interrupt is generated if the eof_irq bit is set for the current stream and a filtering operation has been completed with a match.

- DVD mode

  An eof interrupt is generated if the eof_irq bit is set for the stream 0 and the DMA engine has been initialized with the parameters stored in FRAM.

#### IRQ Processing

Normal processing continues.

- DSS/DVB mode

  The eof_flag is set along with the **STREAM_NUMBER** in the status word.

  The values of target_match and other_match are stored in the status word which is pushed into the **LINK_STAT_FIFO**.

  This information can be useful for the application software.

- DVD mode

  The eof_flag is set along with the **STREAM_NUMBER** in the status word.

### 16.7.6 End of Section Transfer

#### IRQ Generation

- DSS/DVB mode

  An eos interrupt is generated if the eos_irq bit is set for the current stream number and a section has been completely transferred by the DMA controller to memory.

- DVD mode

  An eos interrupt is generated if the eos_irq bit is set for the stream 0 and the DMA engine has finished the transfer of a packet.

#### IRQ Processing

Normal processing continues.

- DSS/DVB mode

  The eos_flag is set along with the **STREAM_NUMBER** in the status word. The status word is written into the **LINK_STAT_FIFO**.

- DVD mode

  The eos_flag is set along with the **STREAM_NUMBER** in the status word.

### 16.7.7 Incomplete Filtering at End of Packet

#### IRQ Generation

An incomplete filtering interrupt is generated if the incomplete_irq bit is set, a section filtering operation is not complete at the end of a packet and at least one temporary match is pending.

The filtering is considered as successful and the section transfer starts.

#### IRQ Processing

Normal processing continues.

The values of **TARGET_MATCH** and **OTHER_MATCH** are stored in the status word and the **INCOMPLETE_FILTER** bit is set along with the **STREAM_NUMBER** and **FILTER_OFFSET** in the status word and pushed into the **LINK_STAT_FIFO**.

The application software can use this information to complete the section filtering.

### 16.7.8 Adaptation Field reception

#### IRQ Generation

An AF interrupt is generated if:

- the **AF_IRQ** bit is set for the current stream number and
- an AF with at least one flag set is present in the packet and
- the previous one has been read by the CPU.

The first 8 bytes of the AF are stored into the **AF_REG1-0**. If the AF is less than 8 bytes, some payload bytes are written in the AF buffer. If the AF length is zero, there is no match and no storage.

#### IRQ Processing

Normal processing continues. Once Adaptation Field information has been written into the **AF_REG** registers, PCR counters value is latched and no new AF information can overwrite it until the CPU has read the register **AF_REG1**. If another AF is received before the previous one has been read by the CPU, it is simply discarded. The **AF** flag is set in the status word with the **STREAM_NUMBER** and an interrupt is generated.

### 16.7.9 SDAV Underflow Error

#### IRQ Generation

A SDAV underflow is generated if a SDAV packet has started to be output to the SDAV port and no more data is present in the SDAV block to be sent.

#### IRQ Processing

The stream is not disabled. The output packet is corrupted and further data belonging to that packet is discarded. Since the SDAV operates independently from the rest of the Transport Stream Demultiplexor, the **STREAM_NUMBER** present in the status word may not be relevant. When such a SDAV underflow error occurs, the **SDAV_UNDERFLOW** bit is set along with the **STREAM_NUMBER** in the status word and the status word is written into the **LINK_STAT_FIFO**.

### 16.7.10SDAV Overflow Error

#### IRQ Generation

A SDAV underflow is generated if the SDAV buffer is full and new data is presented at the input of the buffer to be stored.

#### IRQ Processing

The stream is not disabled.

Meanwhile, a new packet has possibly started being processed by the Transport Stream Demultiplexor. Note that the SDAV block is supposed to accept data as delivered by the rest of the Transport Stream Demultiplexor and can in no way suspend Transport Stream Demultiplexor operation. When this occurs, the output of the current packet is aborted and all remaining data belonging to that packet is discarded. The read and write pointers are reset. If a new packet_start is already present, data input is not stopped and overwrites the discarded data. The read pointer points now to the first byte of this new packet.

Since the SDAV operates independently from the rest of the Transport Stream Demultiplexor, the **STREAM_NUMBER** present in the status word may not be relevant. When such a SDAV overflow error occurs, the **SDAV_OVERFLOW** bit is set along with the **STREAM_NUMBER** in the status word and the status word is written into the **LINK_STAT_FIFO**.

## 16.8 Memory and register map

The total address space for the Transport Stream Demultiplexor block is 4 kbyte, arranged as1024 words of 32 bits. The block is in the peripheral space of the memory map, with base address *TransportDemuxBase*, whose value is given in the Chapter 8. All addresses in this chapter are hexadecimal byte offsets from *TransportDemuxBase* in the range 0x000 - 0xFFF.

All the resources are accessed as 32-bit words and all the addresses point to bytes.

### 16.8.1 Global address map

Figure 16.20 and Table 16.20 show the locations of the FRAM and registers in the Transport Stream Demultiplexor block.

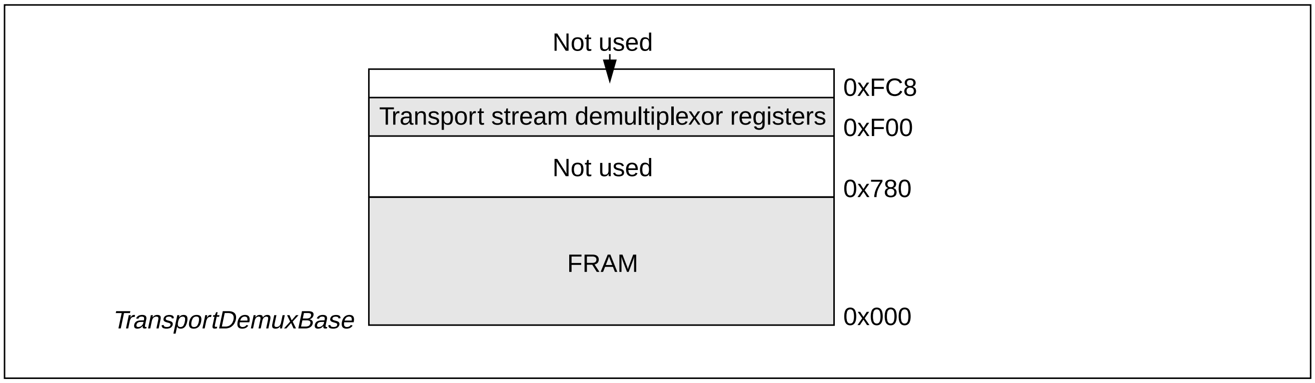


Figure 16.20 Transport stream demultiplexor address map

| Address range | Size | Resource | Type |
|---|---|---|---|
| 0x000 - 0x77C | 480 Words | FRAM | R/W |
| 0x780 - 0xEFC | 480 Words | Not Used | |
| 0xF00 - 0xFC4 | 51 Words | Registers | R/W |
| 0xFC8 - 0xFFC | 15 Words | Not used | |

Table 16.20 Transport stream demultiplexor address map

### 16.8.2 FRAM contents

| Address | Size | Resource | Type |
|---|---|---|---|
| 0x000 - 0x4FC | 320 words | Filter data | R/W |
| 0x500 - 0x57C | 32 words | Descrambling keys | R/W |
| 0x580 - 0x5F0 | 32 words | Stream configuration #1 | R/W |
| 0x600 - 0x67C | 32 words | Stream configuration #2 | R/W |
| 0x680 - 0x6FC | 32 words | Stream configuration #3 | R/W |
| 0x700 - 0x77C | 32 words | IRQ registers | R/W |

Table 16.21 FRAM address map

#### Descrambling keys

Eight different descrambling key sets are stored in FRAM, numbered 0 to 7. Each key set contains 2 keys of 64 bits each. They are mapped as following:

| Address | Key set | Key Parity | Key bits | Type |
|---|---|---|---|---|
| 0x500 | 0 | Even | 31:0 (LSW) | R/W |
| 0x504 | | Even | 63:32 (MSW) | R/W |
| 0x508 | | Odd | 31:0 (LSW) | R/W |
| 0x50C | | Odd | 63:32 (MSW) | R/W |
| 0x510 | 1 | Even | 31:0 (LSW) | R/W |
| 0x514 | | Even | 63:32 (MSW) | R/W |
| 0x518 | | Odd | 31:0 (LSW) | R/W |
| 0x51C | | Odd | 63:32 (MSW) | R/W |
| ... | ... | ... | ... | ... |
| 0x578 | 7 | Odd | 31:0 (LSW) | R/W |
| 0x57C | | Odd | 63:32 (MSW) | R/W |

Table 16.22 Descrambling keys

#### Stream configuration words

The tables in this section define the stream configuration words **Stream_conf_1-3**.

**Stream_conf_1** contains the filter configuration, the behavior for each stream and the interrupt generation scheme, as shown in Table 16.23. **SEC_F_INV_REG** cannot be loaded with 111. The interrupts are generated if mask = 1.

| Bit | Signal Name | Comment | Type |
|---|---|---|---|
| 31 | EOF_IRQ | DVB IRQ Mask on end of Section Filtering<br>DSS End of Conditional Filtering<br>DVD Configuration Has Been Loaded | R/W |
| 30 | EOS_IRQ | DVB IRQ Mask on end of Section<br>DSS End of Packet<br>DVD End of Sector | R/W |
| 29 | AF_IRQ | DVB IRQ Mask on af IRQ | R/W |
| 28 | OUTPUT_PACKET | 1 Send Packet to SDAV<br>0 Ignored by SDAV | R/W |
| 27 | STREAM_TO_BUFFER | 1 Enable Transfer to DMA Buffer<br>0 Bytes are Only Read for SDAV | R/W |
| 26 | ERROR_PATT | 1 Insert Error Pattern on CC Error<br>0 No Insertion of Error Code | R/W |
| 25 | PES_DES_EN | 1 Enables PES Level Descrambling<br>0 No PES Level Descrambling | R/W |
| 24 | PES_NSEC | 1 Stream PES<br>0 Stream SECTION | R/W |
| 23 | SEC_F_INV | 1 Not Equal Mode<br>0 Equal Mode | R/W |
| 22:20 | SEC_F_INV_REG | Offset in filter for ‘Not Equal’ Filter | R/W |
| 19:16 | FILTER_LENGTH | Size of filters | R/W |
| 15:10 | FILTER_NB | Number of filters | R/W |
| 9:0 | FRAM_ADDRESS | Filter start address in FRAM | R/W |

Table 16.23 Stream configuration word **Stream_conf_1**

| Bit | Signal Name | Comment | Type |
|---|---|---|---|
| 31 | INCREMENT | 1 DMA increment (memory)<br>0 no DMA increment (MPEG) | R/W |
| 30:28 | BUFFER_SIZE | DMA circular buffer size | R/W |
| 27:18 | STOP_ADDR | DMA stop address(14:5) | R/W |
| 17:11 | START_ADDR | DMA start address(14:8) | R/W |
| 10:7 | DMA_BANK_ADDR | DMA address(31:28) | R/W |
| 6:0 | DMA_HIGH_ADDR | DMA address(21:15) | R/W |

Table 16.24 Stream configuration word **Stream_conf_2**

**Stream_conf_3** is defined in Table 16.25. This word contains the DMA low address for each stream.

| Bit | Signal Name | Comment | Type |
|---|---|---|---|
| 31:28 | CC_COUNTER_IN | Current Continuity Counter Value | R/W |
| 27 | SUSPEND | DVB - PES:<br>1 Bypass first CC check<br>0 CC check takes place<br>DVB - SEC:<br>1 Section cut<br>0 No section cut<br>DSS:<br>1 Hunt mode<br>0 No hunt mode | R/W |
| 26:12 | DMA_LOW_ADDR | DMA address(14:0) | R/W |
| 11:0 | TRANSFER_LENGTH (XFER_COUNT_REG) | DVB - PES (when PES scrambling): remaining bytes of the PES header if the header exceeds 184 bytes<br>DVB - SEC: remaining bytes in current section<br>DSS: Bit 0 = HD[1] (Toggle bit) | R/W |

Table 16.25 Stream configuration word **Stream_conf_3**

### 16.8.3 Register map

Table 16.26 lists all the Transport Stream Demultiplexor registers with their addresses as offsets from *TransportDemuxBase*. The contents of the registers are given in the *STi5500 Register Manual*.

| Address | Bits | Name | Access | Reset value |
|---|---|---|---|---|
| 0xF00 - 0xF7C | 0:0 | **STREAM_EN_REG[31:0]** | R/W | 0x0 |
| 0xF80 | 20:0 | **LINK_STAT_REG** | R/W | bit 2 = 0 |
| 0xF84 | 31:0 | **LINK_STAT_FIFO** | R | - |
| 0xF88 | 11:0 | **PACKET_LENGTH_REG** | R/W | 0xBC |
| 0xF8C | 5:0 | **TIME_OUT_REG** | R/W | 0x38 |
| 0xF90 | 9:0 | **MODE_REG** | R/W | 0x001 |
| 0xF94 | 5:0 | **PCR_STREAM_REG** | R/W | 0x0 |
| 0xF98 | 31:0 | **AF_REG0** | R | - |
| 0xF9C | 31:0 | **AF_REG1** | R | - |
| 0xFA0 | 31:0 | **V_PTS_REG** | R | - |
| 0xFA4 | 31:0 | **A_PTS_REG** | R | - |
| 0xFA8 | 31:0 | **PCR_REG** | R | - |
| 0xFAC | 8:0 | **PCR_EXT_REG** | R | - |

Table 16.26 Transport stream demultiplexor registers

| Address | Bits | Name | Access | Reset value |
|---|---|---|---|---|
| 0xFB0 | 5:0 | **AR_SIZE_REG** | R/W | 0x3E |
| 0xFB4 | 26:0 | **SDAV_CONF_REG** | R/W | bits 22, 15, 14, 12 = 0 |
| 0xFB8 | 5:0 | **SDAV_DMA_EN_REG** | R/W | bit 0 = 0 |
| 0xFBC | 31:0 | **SDAV_DATA_REG** | R/W | - |
| 0xFC0 | 0:0 | **EN_LINK_REG** | R/W | 0x0 |
| 0xFC8 | 12:0 | **EXTRA_BITS_REG** | R/W | bit(0) = 0 |
| 0xFCC - 0xFFC | Not used | | | |

Table 16.26 Transport stream demultiplexor registers

### 16.8.4 Register contents

The register contents are given in the STi5500 register manual.

## 16.9 Glossary

| | |
|---|---|
| AR | Acquisition RAM |
| ARAM | Acquisition RAM |
| FRAM | Filter RAM |
| IRD | Integrated Receiver Decoder box |
| NRSS | National Renewable Security System |
| PES | Packet Elementary Streams |
| PSI | Program Specific Information |
| SDAV | Simplified Digital Audio Video |
| TP | Transport Packets |

# 17 MPEG DMA controllers

The on-chip MPEG Audio and MPEG Video Decoders are memory-mapped, and contain Compressed Data (CD) FIFOs for audio (in the Audio Decoder) and one each for video and sub-picture (in the Video Decoder). Some applications require data to be transferred to these FIFOs using a dedicated MPEG DMA controller.

There are two such MPEG DMA controllers MPEGDMA0-1 available on the STi5500, which are time-shared by the three CD FIFOs. MPEGDMA0-1 can each be configured for transferring blocks of compressed data to any of the FIFOs. The DMA will remain dedicated to that FIFO until the complete block of data has been sent (i.e. cannot be interrupted).

A third MPEG DMA controller MPEGDMA2 is provided for transferring data to the SDAV interface of the Hardware Transport Stream Demultiplexor. MPEGDMA2 must be configured to send to the SDAV interface.

The MPEG DMA transfer is initiated by the CPU, using a channel as described in Appendix A. Control registers can be set to define the characteristics of each DMA transfer burst in response to a request, or to suspend the transfer. The base address for the output buffer in the memory space and the size of transfer in bytes are set by the *out* (output) instruction from the CPU to the MPEG DMA controller channel. For channel mapping refer to the Memory Map.

## 17.1 MPEG DMA transfers

To perform a DMA transfer to an MPEG decoder, the MPEG DMA controller must first be initialized and then an output to the MPEG DMA channel be executed by the CPU.

The control registers are described in section 17.2.

The **MPEGBurstSize** register controls the number of bytes transferred each time the DMA controller samples the **notCDREQ** signal active. This should be programmed with a burst size appropriate for the MPEG decoder fifo.

After sampling the **notCDREQ** signal active the signal is ignored until the burst size in bytes has been transferred, the last write cycle of the burst has completed, and the hold-off time programmed in the **MPEGHoldoff** register in cycles has expired from the last write cycle completion**.** If the **notCDREQ** signal is active after this time then the DMA controller will transfer another burst of data.

The **MPEGSuspend** register bit must be set to ‘1’ before a transfer is initiated, otherwise the transfer will not start.

The **MPEGBurstSize** and **MPEGHoldoff** registers are not altered by transfer operations and only have to be reset when changing to another decoder.

The final stage of initializing the DMA transfer is to execute an output to the **MPEGDMA** channel which sets up the source base address and the DMA transfer size. This also deschedules the software until the transfer is complete.

The maximum transfer size is 65535 bytes.

The DMA module will only transfer data when the appropriate **notCDREQ** input is active after the output to the DMA channel. The DMA then transfers the programmed burst size in bytes of data to the location set for the FIFO buffer in memory. Note, if there are less than **BurstSize** bytes left to transfer then only these bytes will be transferred.The FIFO buffer address is *not* incremented.

The MPEG DMA controller fetches words from the source address whenever possible and buffers these to perform word writes to the destination address whenever possible.

During a transfer DMA operations can be suspended by setting the **MPEGSuspend** register bit to a ‘0’. Note that although no new write transfers will be started after this bit has been set to ‘0’, software must wait for a time long enough for the current write transfer to finish before assuming that no DMA writes are being performed. Transfers will start again when the **MPEGSuspend** register bit is set to ‘1’.

When the number of bytes programmed in the *out* instruction have been transferred the channel output is acknowledged to the CPU and the software rescheduled.

The destination address for the data is programmable in real time in the MPEG DMA0-1 controller registers, and thus can be directed to any of the three decoder FIFOs via the **MPEGDecoderSel** register. For MPEGDMA2 this register must be set to send to the SDAV interface.

The register base addresses for the MPEG DMA controllers are given in the *Memory Map* chapter.

## 17.2 MPEG control registers

| **MPEGBurstSize** | **MPEGDMA base address + #00** | **Write only** |
|---|---|---|
| **Bit** | **Bit field** | **Function** |
| 4:0 | **BurstSize4:0** | DMA transfer burst size in response to **notCDREQ0-3.**<br>**BurstSize4:0 Transfer**<br>00000 32 bytes per burst<br>00001 1 byte per burst<br>00010 2 bytes per burst<br>... ...<br>11111 31 bytes per burst |
| 7:5 | | RESERVED. Write 0 |

Table 17.1 **MPEGBurstSize** register format

| **MPEGHoldoff** | **MPEGDMA base address + #04** | **Write only** |
|---|---|---|
| **Bit** | **Bit field** | **Function** |
| 4:0 | **Holdoff4:0** | DMA transfer holdoff time from the end of one burst to re-sampling **notCDREQ0-3**.<br>**Holdoff4:0 Holdoff time in system clock cycles**<br>00000 32 cycles<br>00001 1 cycle<br>00010 2 cycles<br>... ...<br>11111 31 cycles |
| 7:5 | | RESERVED. Write 0 |

Table 17.2 **MPEGHoldoff** register format

| **MPEGSuspend** | **MPEGDMA base address + #08** | **Write only** |
|---|---|---|
| **Bit** | **Bit field** | **Function** |
| 0 | **Suspend** | Suspend DMA operations<br>0 suspend DMA<br>1 enable DMA (normal operation) |
| 7:1 | | RESERVED. Write 0 |

Table 17.3 **MPEGSuspend** register format

The meaning of the **MPEGDecoderSel** register differs between the MPEGDMA controllers, since the outputs are connected to different destinations.

| **MPEGDecoderSel** | **MPEGDMA base address + #0C** | **Write only** |
|---|---|---|
| **Bit** | **Bit field** | **Function** |
| 1:0 | **DecoderSelect1:0** | Select Decoder (controls polling of **notCDREQ** and FIFO base address) for DMA transfer<br>00 select **notCDREQ0**<br>01 select **notCDREQ1**<br>10 select **notCDREQ2**<br>11 select **notCDREQ3** |
| 7:2 | | RESERVED. Write 0 |

Table 17.4 **MPEGDecoderSel** register format for MPEGDMA0-1

On STi5500 for MPEGDMA0-1, the hardware is configured such that the Decoder Select values correspond to the MPEG CD FIFOs given in Table 17.5.

| **MPEGDecoderSel** | **Request Signal** | **FIFO buffer address** | **MPEG module** |
|---|---|---|---|
| 00 | **notCDREQ0** | #00001800 | Video |
| 01 | **notCDREQ1** | #00001A00 | Audio |
| 10 | **notCDREQ2** | #00001C00 | Sub-picture |
| 11 | **notCDREQ3** | #00001E00 | not defined |

Table 17.5 MPEG modules and write addresses

| **MPEGDecoderSel** | **MPEGDMA base address + #0C** | **Write only** |
|---|---|---|
| **Bit** | **Bit field** | **Function** |
| 1:0 | **SDAVSelect1:0** | Select Destination for DMA transfer<br>00 Reserved<br>01 Reserved<br>10 Reserved<br>11 SDAV |
| 7:2 | | RESERVED. Write 0 |

Table 17.6 **MPEGDecoderSel** register format for MPEGDMA2

# 18 MPEG video decoder

This chapter describes the STi5500 MPEG video decoder. The video decoder decompresses a MPEG 2 bit-stream and constructs a picture. The registers to control the decoder are described in the *STi5500 Register Manual* and the display functions are described in Chapter 21.

## 18.1 Decoder operation

The video decoder is a picture decoder; it decodes a whole picture and then stops until instructed to decode the next picture present in the video bit-stream.

Normally, the decoding of a new picture commences in response to the start of display of a new picture. The registers whose contents can change from picture to picture are double-banked and are updated automatically when decoding starts. The bit-stream is read from the bit buffer into the variable-length code decoder (VLD), and picture reconstruction can commence. Any predictors required are fetched from the appropriate area of the external memory, and the reconstructed picture is written back into the area of this memory assigned to the decoded picture.

While a picture is being decoded the start code detector is used to locate the start of the next picture header, which the CPU then reads in order to set up the double-banked registers for the decoding of the next picture.

All of these tasks can be synchronized using interrupts generated on start code hits and vertical sync signals.

### 18.1.1 Start code search

The video decoder is able to decode in its entirety a video bit-stream from the slice layer downwards. The higher layers (i.e. picture and upwards) are decoded by the driver in order to extract the information needed for decoding and set up the appropriate video decoder registers and quantization tables. Since the header information is byte-aligned and requires minimal interpretation, this task represents only a small load on the CPU.

The start code detector parses the bit-stream stored in the bit buffer and locates start codes corresponding to picture layer and above. When one of these start codes has been found, the start code detector stops and raises an interrupt.

The CPU is then able to read the header data following the start code. The start code detector starts automatically whenever the decoding of a new picture starts and on user command. In normal operation, start code parsing is performed one picture in advance of decoding.

### 18.1.2 Operation in bandwidth reduction mode

In bandwidth reduction mode the decoder requires the use of three frame buffers in external memory. This is the normal mode of operation, where I, P and B-frames are decoded into and displayed from frame buffers in external memory. This mode is highly optimal in terms of memory bandwidth usage.

### 18.1.3 Operation in reduced memory mode

In reduced memory mode the decoder requires the use of only two frame buffers in external memory. The bidirectional frames are decoded and displayed on-the-fly.

As the display is interlaced the B-frames must be decoded twice, once for each field. This requires the decoder to be able to loop back in the bit-buffer and re-decode an image.

This is controlled by the driver in conjunction with the start code detector and is described further in section 18.4.1.

## 18.2 Resets

Hard reset is a global signal and is described in the *System Services* chapter.

The following types of soft reset can be used for the video decoder:

- A *total soft reset* is generated by setting and resetting bit **VID_CTL.SRS** and register **AUD_RES**. They must be set for a duration of at least 540ns.
- The sub-picture subsystem may be *soft reset* by setting and resetting **VID_CTL.SPR**.
- A *pipeline reset* is generated by setting and resetting bit **VID_CTL.PRS**. It must be set for a duration of at least 40ns.

After a soft reset, all processes concerning decoding and bit buffer control are reset. Any data remaining in the bit buffer, the compressed data FIFO and the start code detector FIFO are lost.

The interrupt unit is reset. All registers maintain their contents and the display process is not disturbed. A soft reset would normally be used when the decoding of the current bit-stream must be terminated and it is required to restart on a new sequence.

After a hard or a soft reset or a video soft reset, the first task performed by the pipeline when it has been enabled will always be a search for the beginning of a new sequence. The bit buffer data is flushed until the first picture start code following a sequence start code is detected by the pipeline, at which time it stops. At this point normal picture decoding behavior is resumed. After a hard or a soft reset, the first search performed by the start code detector in response to the first DSYNC will always be a search for a sequence start code, after which it stops. After this, the start code detector operates normally.

A pipeline reset terminates the decoding of the current picture. The remaining bits of the picture are flushed from the bit buffer until the next picture start code is detected by the pipeline. At this point normal behavior is resumed, i.e. the pipeline waits for the next picture decoding instruction. No other part of the circuit is affected by a pipeline reset. A pipeline reset would normally be used as part of a manual error recovery procedure. A pipeline reset has no effect if the decoding pipeline is in its idle state.

## 18.3 Bit buffer and start code detection (video)

### 18.3.1 Bit buffer

The transfer of compressed data is carried out using the MPEG DMA engines described in Chapter 17. Compressed data can be taken from any memory space visible to the CPU and transferred to the relevant elementary stream decoder.

### 18.3.2 Start code detection

The start code detector operates in parallel with the decoding pipeline. The purpose of this unit is to allow external access to the header data which follows start codes in the input bit-stream. Compressed data is read twice from the bit buffer- once into the pipeline, and once into the start code detector through the 128-byte header FIFO. The transfer of data into the header FIFO does not affect the bit buffer level; only the data transfer into the pipeline can reduce the bit buffer level.

Start code detection is initiated in two ways:

- Automatically whenever the internal event DSYNC occurs. DSYNC is derived from VSYNC as described in section 18.8.1. A DSYNC is generated every time the pipeline starts a new picture decoding task.
- By software writing to the **VID_HDS** register with bit **VID_HDS.HDS** set.

When start code detection has been started, data is read continuously from the bit buffer into the header FIFO and parsed by the start code detector, which receives the FIFO output data. When a start code is detected, the data scanning stops and the status bit **VID_STA.SCH** becomes 1. When a start code has been detected, it can be identified by reading the **VID_HDF** register. The start code detector detects all start codes other than the codes from 0x00000102 through to 0x000001AF. The first slice start code 0x00000101 can be optionally detected to help driver development.

The register **VID_HDF** should always be read twice to return a 16-bit value. The most significant byte is read first. After detection of a start code, **VID_HDF** will return one of the 16-bit values shown in Figure 18.1.

| | First read | Second read | Third read | |
|---|---|---|---|---|
| **VID_HDF** | Last byte of Start Code | First header byte | Header data | |
| **VID_HDF** | 01 | Last byte of Start Code | First header byte | |

Figure 18.1 States of **VID_HDF** after detection of a start code

The first step is to examine the first byte read from **VID_HDF**. If this contains 0x01, then the start code can be identified by a second read at the same address. If the first byte is not 0x01 then it must be the last byte of the start code and the second byte is the first byte of the header data. In both cases subsequent reads from **VID_HDF** will give access to the header data which follows the start code.

Scanning for start codes will recommence on the next DSYNC or a write to **VID_HDS.HDS**. *Whenever a start code has been detected, the* **VID_HDF** *register must be read in order for the start code detector to restart correctly*. The number of reads before a manual or automatic (DSYNC) restart must always be even.

The first start code search after a hard or soft reset will be a search for a sequence header start code; all other start codes will be ignored. When this start code has been read, all subsequent searches will look for any start codes other than slice start codes.

The two status bits **VID_STA.HFE** (header FIFO empty) and **VID_STA.HFF** (header FIFO full) indicate the state of the header FIFO. Reading from HDF must never be performed if **VID_STA.HFE** is 1. **VID_STA.HFF** is set whenever the header FIFO contains at least 66 bytes.

The start code detector can also be programmed to stop on the first slice of the picture. This allows the use of the start code search even after reception of the picture start code. All header data that is not used by the application can then be skipped without risk, in order to jump to the next picture start code.

This mode is enabled by setting bit **VID_HDS.SOS**. To differentiate between first slice start code (00 00 01 01) and other start codes, it is possible to detect at which position (MSB or LSB) the Last Byte of Start code is positioned in the **VID_HDF** register. Register bit **VID_HDS.SCM** when set indicates that the Last Byte of Start code is held by the MSB of **VID_HDF**; it is zero otherwise.

## 18.4 Video decoding pipeline control

The pipeline is the core of the decoder. It is that part of the circuit which converts the compressed bit-stream data for each picture into a decoded (or reconstructed) picture. These pictures can be frame or field pictures. The operation of the pipeline is controlled picture-by-picture. The decoding of a new picture can potentially start on every VSYNC, but usually the rate of decoding is faster than the VSYNC rate.

The pipeline is controlled by the pipeline controller. When the pipeline controller starts the decoding pipeline a DSYNC signal is issued and **VID_STA.PSD** is set.

This signal is also sent to the start code detector. When the pipeline has completed its decoding operation, a completion signal is sent to the pipeline controller, which is then able to launch another decoding operation, either immediately or when the next VSYNC occurs.

The pipeline controller interprets certain bits of the decoding instruction, which must be set up by the user before the start of each new task. The remaining bits of the instruction define the decoding task itself.

The pipeline receives its compressed data from the bit buffer. This data is first processed by the variable length decoder (VLD) which regenerates the run/level coded DCT coefficients and the motion vectors (if present) for each macroblock. The picture data is reconstructed by passing the run/level data through the inverse quantizer and inverse DCT blocks.

This is then added to the predictors which have been fetched from the memory taking into account the macroblock prediction modes and motion vectors.

Finally, the decoded picture is written back into the memory, from where it can be accessed by the display unit for output.

The pipeline is also able to skip through picture data for various reasons. The different possibilities are:

- Skip to Next Sequence. This occurs unconditionally on the first instruction execution after a hard or soft reset (see section 18.2). Compressed data is skipped until the first picture start code following a sequence start code is found. The pipeline then indicates task completion and waits for a new instruction.
- Skip to Next Picture. This occurs either after a pipeline reset (see section 18.2) or when the decoding instruction specifies that one or two pictures should be skipped (see section 18.8.1). In the first case compressed data is skipped until the next picture start code is found, after which the pipeline indicates task completion and waits for a new instruction. In the second case, after the skipping operation the decoding of the following picture is started immediately.
- Skip to Next Slice. This occurs after automatic error concealment (see section 18.8.2). Compressed data is skipped until the next slice start code in the picture is found, after which normal decoding resumes.

Before starting to decode a sequence, certain static parameters must be set up. These are:

- MPEG-1 or MPEG-2 mode selection. Bit **VID_TIS.MP2** must be set for an MPEG-2 sequence, reset for an MPEG-1 sequence.
- Decoded picture size. Register **VID_DFW** must be set up with the picture width in macroblocks, and register **VID_DFS** must be set up with the number of macroblocks in the picture.

Decoding is enabled by setting bit **VID_CTL.EDC**.

### 18.4.1 Decoder / display sequencing

The decoder has two main modes of operation:

- Bandwidth reduction mode (normal mode)
- Memory reduction mode

The modes can be selected on a frame by frame basis. The user can decide for each frame how it is decoded by setting or clearing the bit **OTF** in the **VID_PPR1** and **FLY** in **VID_DCF** for each picture. The bit **FNF** in **VID_DCF** must also be correctly set depending on the type of picture to be decoded. If a picture is decoded on-the-fly then, in general, it has to be decoded twice to allow the display of both fields. To decode the picture twice, the bit **DC2** in **VID_TRF** has to be set and 1 must be written to the register **VID_DC2** during the first decode so that the pipeline will loop back to the start of the frame as soon as the first decode is complete.

In normal usage the decoder will be configured in one mode or the other at the start of decode. This is important because the phasing of the decoding and display processes is not the same in the two modes. Figure 18.2 shows the required phasing for each of the decode modes. It can clearly be seen that there is a full-field phase difference between the two display sequences this makes switching between the two modes of decode during decoding difficult.

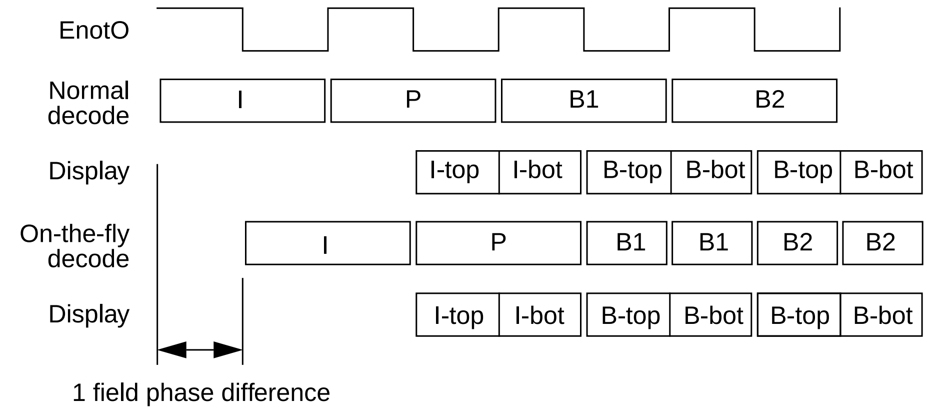


Figure 18.2 Phasing required for decoder modes

If memory reduction mode is selected then the main difference in the decoder control sequencing comes in the decoding of the B-frames. When the start code detector stops on a picture and the picture type is identified as a B-frame then the current position of the bit-buffer read pointer must be stored so that the pipeline can loop back to the beginning of the B-frame to re-decode the picture for display of the second field. The current position of the bit-buffer pointer can be stored by setting then resetting bit 0 in the **VID_LDP** register while the start code detector is aligned at the start of the picture required to be re-decoded. At the same time that this is carried out the temporal reference of the image to be re-decoded must be programmed in **VID_TRF**. This is important because the resolution of the jump back is large and may jump back to a preceding picture therefore the temporal reference is required to make sure the correct picture is re-decoded.

## 18.5 Quantization table loading

The two quantization matrices (intra and non-intra) used by the inverse quantizer must be initialized by the user. There are no built-in quantization matrices. Therefore, they must be loaded either with default matrices or with those extracted from the bit-stream by the ST20.

The quantization tables are double-buffered. This enables one or both tables to be updated without disturbing the decoding task in progress.

The video decoder maintains two bits which record whether one or both of the tables have been modified. A modified table is automatically brought into operation at the start of the next decoding operation, i.e. when the next DSYNC occurs.

After a hard reset, the same pair of tables is always selected. The data previously loaded into the tables is not affected. Other types of reset have no effect on the quantization tables.

The quantization tables are written at the address held in the register **VID_QMW**. Bit **VID_HDS.QMI** is used to select the Intra or Non-Intra quantization table; when it is set, the Intra table is selected; when clear the Non Intra table is selected.

## 18.6 Memory mapping of data

### 18.6.1 Video decoder memory (SDRAM) addressing

The locations in an SDRAM are addressed row by row, bank A then bank B, as shown in Figure 18.3.

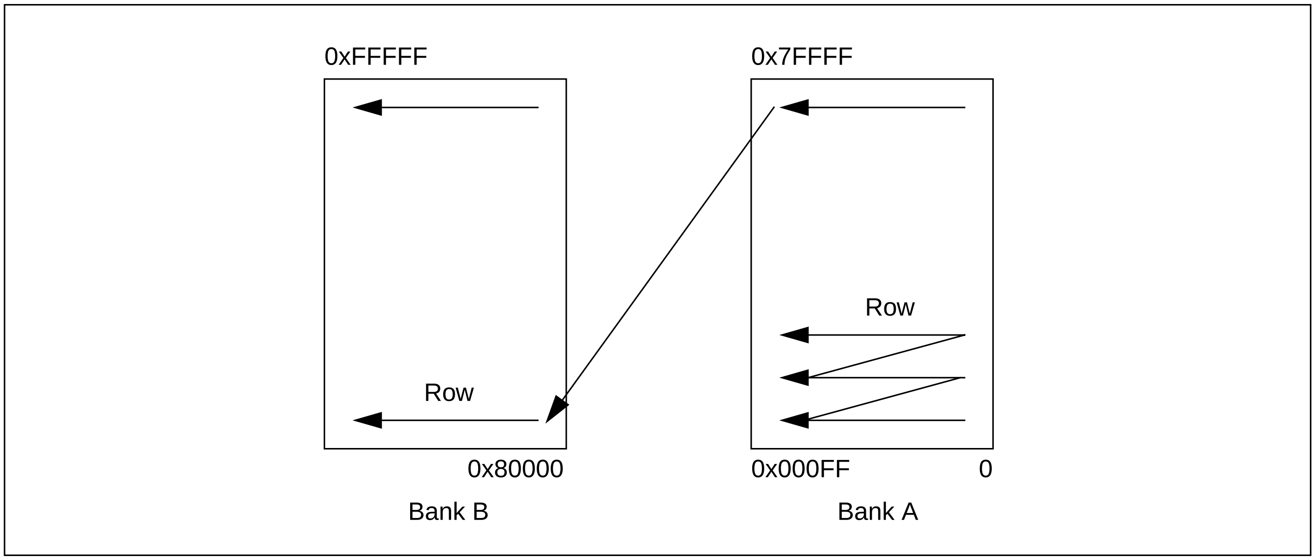


Figure 18.3 Standard addressing in a SDRAM (16-bit words)

### 18.6.2 32-bit word addressing for the CPU

The CPU accesses the SDRAM by using a 19-bit address for each 32-bit word. It is the task of the SDRAM memory controller to remap the logical address space of the CPU onto the SDRAM address space.

The logical address map seen by the CPU is different from the one described in section 18.6.1. For each row, both banks are used. The addresses seen by the CPU through the SDRAM interface are counted in the following order:

Bank A, row0

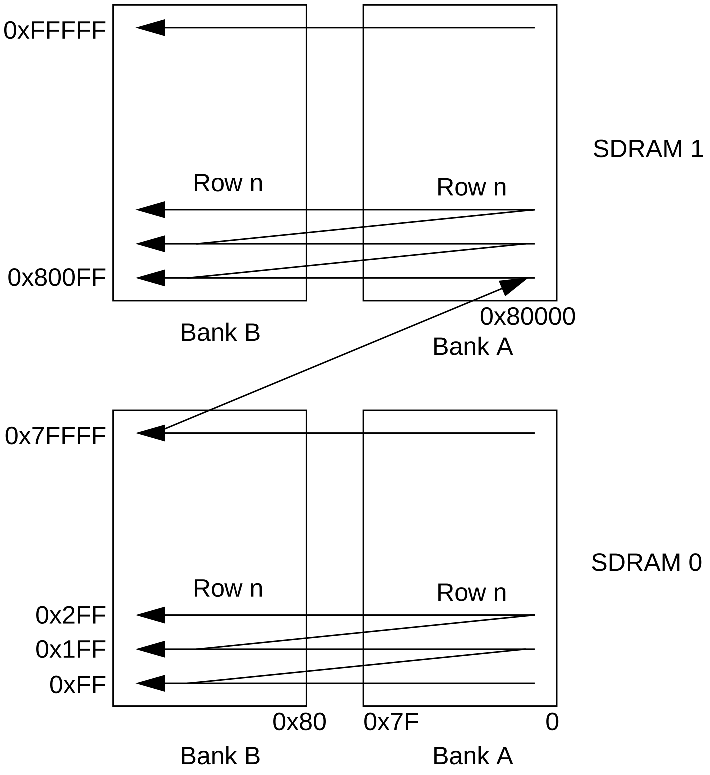
Bank B, row0

Bank A, row1

Bank B, row1

etc...

When using a second SDRAM chip, addresses continue in a similar way, starting from the next address above the first SDRAM. A maximum of two SDRAM chips is supported. This is shown in Figure 18.4.



The system supports up to 2 SDRAM chips

Figure 18.4 32-bit word addressing, as seen by the CPU

### 18.6.3 64-bit word addressing for FIFO processes

The video decoder uses circular buffers mapped into external SDRAM which act as software FIFOs. The processes pertaining to these circular buffers are managed with a 64-bit granularity. The memory mapping for these buffers is similar to that of the CPU and is shown in Figure 18.5.

When using a second SDRAM chip, addresses continue in a similar way, starting from the next address above the first SDRAM. A maximum of two SDRAM chips is supported.

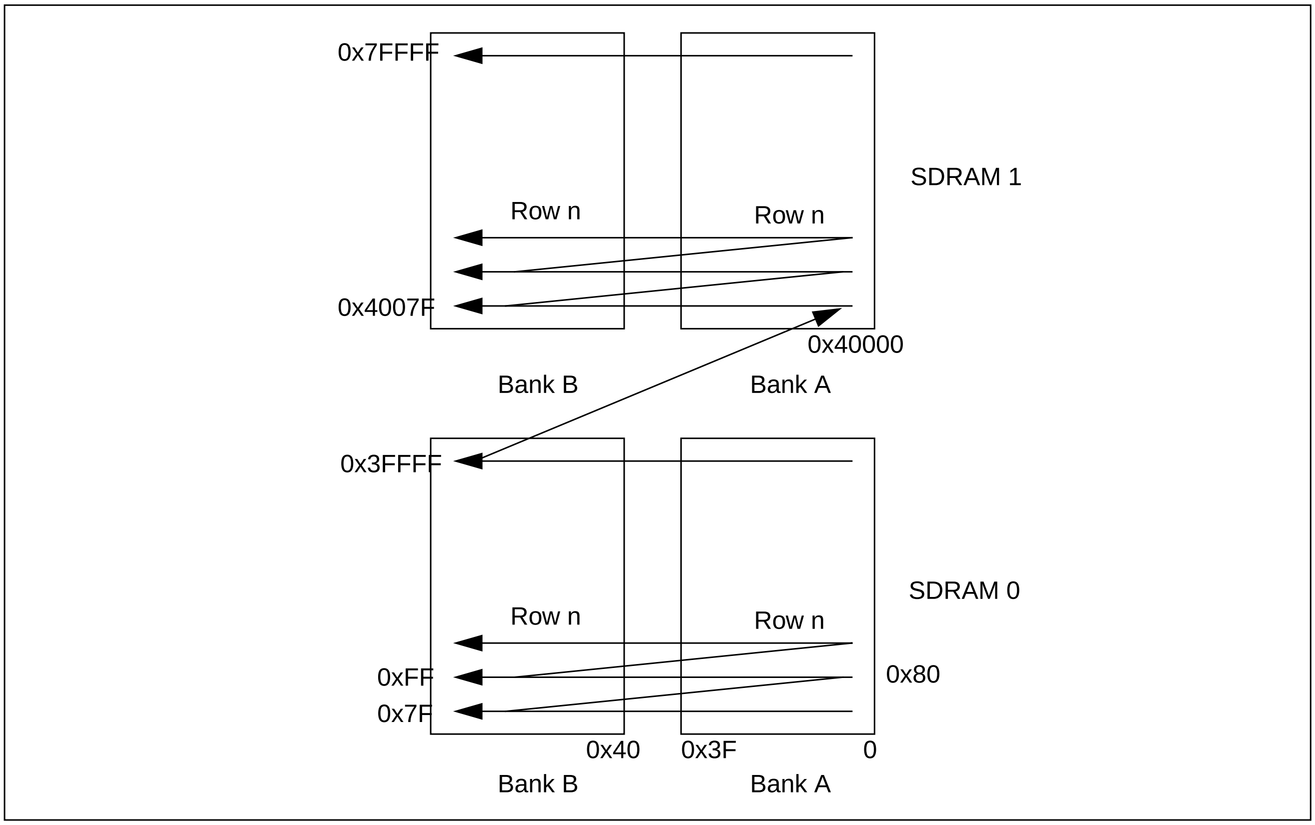


Figure 18.5 64-bit word addressing for FIFO processes

### 18.6.4 Memory segments

The circular buffer start and end pointers are programmed by the user in segments, where each segment is 256 bytes. The values in the configuration registers are numbers of segments. For example a value of 4 means 4 x 256 bytes = 1kbyte or 128 x 64-bit words. This would result in a pointer pointing to a 64-bit word address of 128 (0x80). This address would be physically mapped to the first word in the second row of bank A of SDRAM 0, as shown in Figure 18.6.

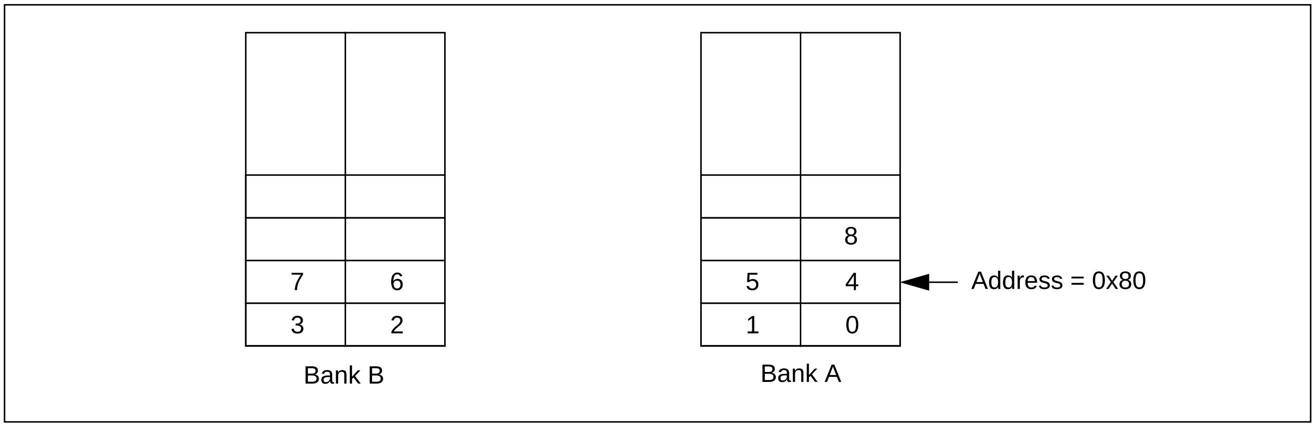


Figure 18.6 SDRAM segments as seen by the user

### 18.6.5 Arrangement of pixel pairs inside a luma SDRAM row

Every SDRAM row in a luma frame contains 256 16-bit words and can store up to two luma macroblocks. Every 16-bit word contains a pair of horizontally adjacent luma pixels. The row itself stores a pair of horizontally adjacent luma macroblocks. The pixel pairs are arranged in line order; the first 16 words store the first line of pixels for the two macroblocks, the next 16 words the second line and so on, as shown in Figure 18.7.

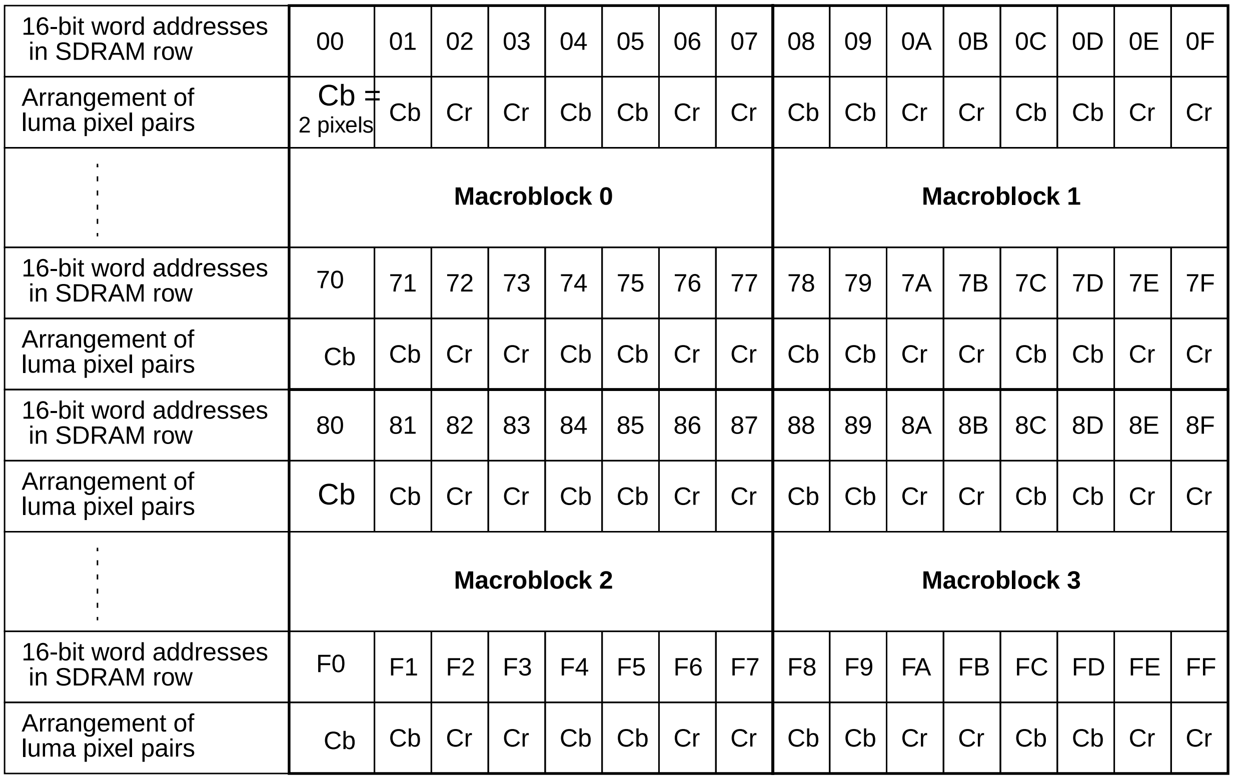
| 16-bit word addresses in SDRAM row | 00 | 01 | 02 | 03 | 04 | 05 | 06 | 07 | 08 | 09 | 0A | 0B | 0C | 0D | 0E | 0F |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Arrangement of luma pixel pairs | Y =<br>2 pixels | Y | Y | Y | Y | Y | Y | Y | Y | Y | Y | Y | Y | Y | Y | Y |
| ⋮ | **Macroblock 0** | | | | | | | | **Macroblock 1** | | | | | | | |
| 16-bit word addresses in SDRAM row | F0 | F1 | F2 | F3 | F4 | F5 | F6 | F7 | F8 | F9 | FA | FB | FC | FD | FE | FF |
| Arrangement of luma pixel pairs | Y | Y | Y | Y | Y | Y | Y | Y | Y | Y | Y | Y | Y | Y | Y | Y |

Figure 18.7 Arrangement of pixel pairs in a luma SDRAM row

### 18.6.6 Arrangement of pixel pairs inside a chroma SDRAM row.

Every SDRAM row in a chroma frame contains 256 16-bit words and can store up to four chroma macroblocks. Every 16-bit word contains a pair of horizontally adjacent 8-bit chroma pixels.

The row stores pixel pairs in line order for macroblocks 0 and 1 and then macroblocks 2 and 3. The Cb and Cr words are interleaved two by two in the linear addressing order, as shown in Figure 18.8.



| 16-bit word addresses in SDRAM row | 00 | 01 | 02 | 03 | 04 | 05 | 06 | 07 | 08 | 09 | 0A | 0B | 0C | 0D | 0E | 0F |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Arrangement of luma pixel pairs | Cb = 2 pixels | Cb | Cr | Cr | Cb | Cb | Cr | Cr | Cb | Cb | Cr | Cr | Cb | Cb | Cr | Cr |
| ⋮ | Macroblock 0 | | | | | | | | Macroblock 1 | | | | | | | |
| 16-bit word addresses in SDRAM row | 70 | 71 | 72 | 73 | 74 | 75 | 76 | 77 | 78 | 79 | 7A | 7B | 7C | 7D | 7E | 7F |
| Arrangement of luma pixel pairs | Cb | Cb | Cr | Cr | Cb | Cb | Cr | Cr | Cb | Cb | Cr | Cr | Cb | Cb | Cr | Cr |
| 16-bit word addresses in SDRAM row | 80 | 81 | 82 | 83 | 84 | 85 | 86 | 87 | 88 | 89 | 8A | 8B | 8C | 8D | 8E | 8F |
| Arrangement of luma pixel pairs | Cb | Cb | Cr | Cr | Cb | Cb | Cr | Cr | Cb | Cb | Cr | Cr | Cb | Cb | Cr | Cr |
| ⋮ | Macroblock 2 | | | | | | | | Macroblock 3 | | | | | | | |
| 16-bit word addresses in SDRAM row | F0 | F1 | F2 | F3 | F4 | F5 | F6 | F7 | F8 | F9 | FA | FB | FC | FD | FE | FF |
| Arrangement of luma pixel pairs | Cb | Cb | Cr | Cr | Cb | Cb | Cr | Cr | Cb | Cb | Cr | Cr | Cb | Cb | Cr | Cr |

Figure 18.8 Arrangement of pixel pairs in a chroma SDRAM row

## 18.7 Using picture pointers

Before the decoding of each picture the following frame buffer pointers must be set up:

- **VID_RFC**, **VID_RFP** - reconstructed frame pointers for chroma and luma;
- **VID_FFC**, **VID_FFP** - forward prediction frame pointers for chroma and luma;
- **VID_BFC**, **VID_BFP** - backward prediction frame pointers for chroma and luma.

A fourth pair of pointers, **VID_DFC**, **VID_DFP**, the displayed frame pointers, is described in section 21.2.1.

**VID_RFP** and **VID_RFC** define the memory buffer to which the decoded picture is written. **VID_FFP**, **VID_FFC**, **VID_BFP** and **VID_BFC** define the areas in memory from which the predictors are fetched.

The rules governing the use of the prediction frame pointers are given below.

Pictures are always stored as frames of interleaved lines, and thus to access a field (top or bottom), the starting address of the frame must be defined.

### P-Frame picture (frame, field or dual-prime prediction)

**VID_FFP** and **VID_FFC** are set to the address of the predictor frame (in which the two predictor fields lie). **VID_BFP** and **VID_BFC** are not used.

### B-Frame picture (frame or field prediction)

**VID_FFP** and **VID_FFC** are set to the address of the forward predictor frame (in which the two predictor fields lie). **VID_BFP** and **VID_BFC** are set to the address of the backward predictor frame (in which the two predictor fields lie).

### P-Field picture (field, 16 x 8 or dual-prime prediction)

When decoding either field, **VID_FFP** and **VID_FFC** are set to the address of the previous decoded I or P frame. **VID_BFP** and **VID_BFC** are not used.

### B-Field picture (field or 16 x 8 prediction)

**VID_FFP** and **VID_FFC** are set to the address of the frame in which the two forward predictor fields lie. **VID_BFP** and **VID_BFC** are set to the address of the frame in which the two backward predictor fields lie.

### I-Pictures

For I-picture decoding, no predictors are necessary, but **VID_FFP** and **VID_FFC** must be set to the address of the last decoded I- or P-picture for use by the automatic error concealment function.

## 18.8 The video pipeline

### 18.8.1 Decoding task control

A task is a single picture decoding operation. A task is specified by the task description or instruction, which is set up before the decoding of each picture. A task commences when the internal signal DSYNC is generated. A task completes (the pipeline becomes idle) when the picture header of the following picture is detected by the pipeline and the picture is entirely reconstructed in the memory. The instruction is double buffered, so that during execution of a decoding task, the instruction for the next task can be set up by the CPU. When the next instruction is activated, a DSYNC can be generated, and the next decoding task started. The buffering mechanism is illustrated in Figure 18.9. Note that some instruction bits are latched by VSYNC, others by a signal from the pipeline controller “new instruction”.

The instruction is written into registers **VID_PPR1** and **VID_PPR2**. If a new instruction is not written, the task descriptor will be the same as the previous one.

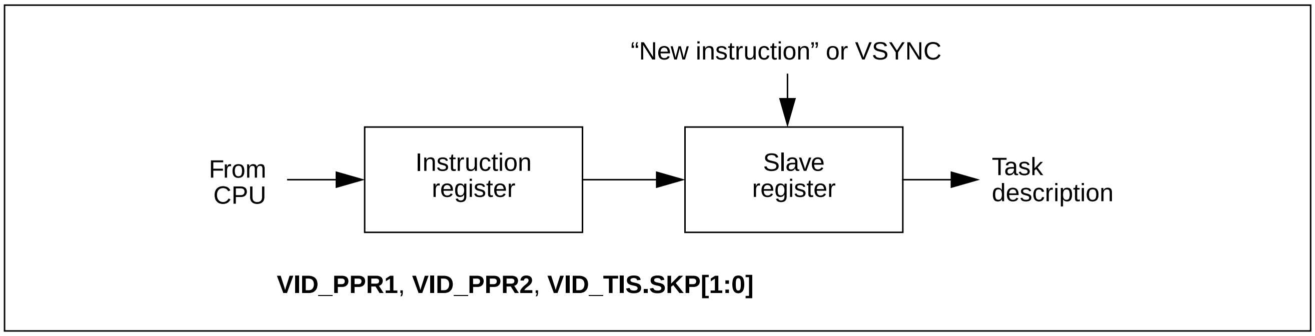

“New instruction” or VSYNC
From CPU
Instruction register
Slave register
Task description
VID_PPR1, VID_PPR2, VID_TIS.SKP[1:0]

Figure 18.9 Instruction buffering

Normally, it is a VSYNC that starts the execution of a new instruction, and thus the generation of DSYNC. If however, a VSYNC occurs before task completion (i.e. before the pipeline becomes idle), the start of the next task will be delayed until the present one is completed. In this way the decoding of a picture can be allowed to extend beyond the nominal period allotted to it, usually one or two VSYNC periods.

Three status bits (and thus interrupts) are associated with pipeline control:

- **VID_STA.PSD** indicates the occurrence of a DSYNC. **VID_STA.PII** indicates that the pipeline is idle.
- **VID_STA.DEI PID** indicates that the decoder is idle, i.e. the pipeline is idle and the next picture start code has been found.

The operation of the pipeline controller is shown in the state diagram of Figure 18.10. The meanings of the abbreviations for the state transitions are given in Table 18.1.

| Abbreviation | Meaning |
|---|---|
| ERC | Automatic error concealment. |
| EXE.FIS | Both VID_TIS.EXE and VID_TIS.FIS are set. |
| EXE.Vsync | Bit VID_TIS.EXE set when external VSYNC occurs. |
| DEI | Pipeline idle interrupt generated. |
| PSC | Picture start code. |
| PSD | Pipeline start decode interrupt generated. |
| SEQ | Sequence start code |
| Skip and decode | VID_TIS = EXE **\|** SKP[01] and VSYNC occurred. |
| Skip and stop | VID_TIS = EXE **\|** SKP[11] and VSYNC occurred. |
| Skip twice and decode | VID_TIS = EXE **\|** SKP[10] and VSYNC occurred. |

Table 18.1 State transition abbreviations

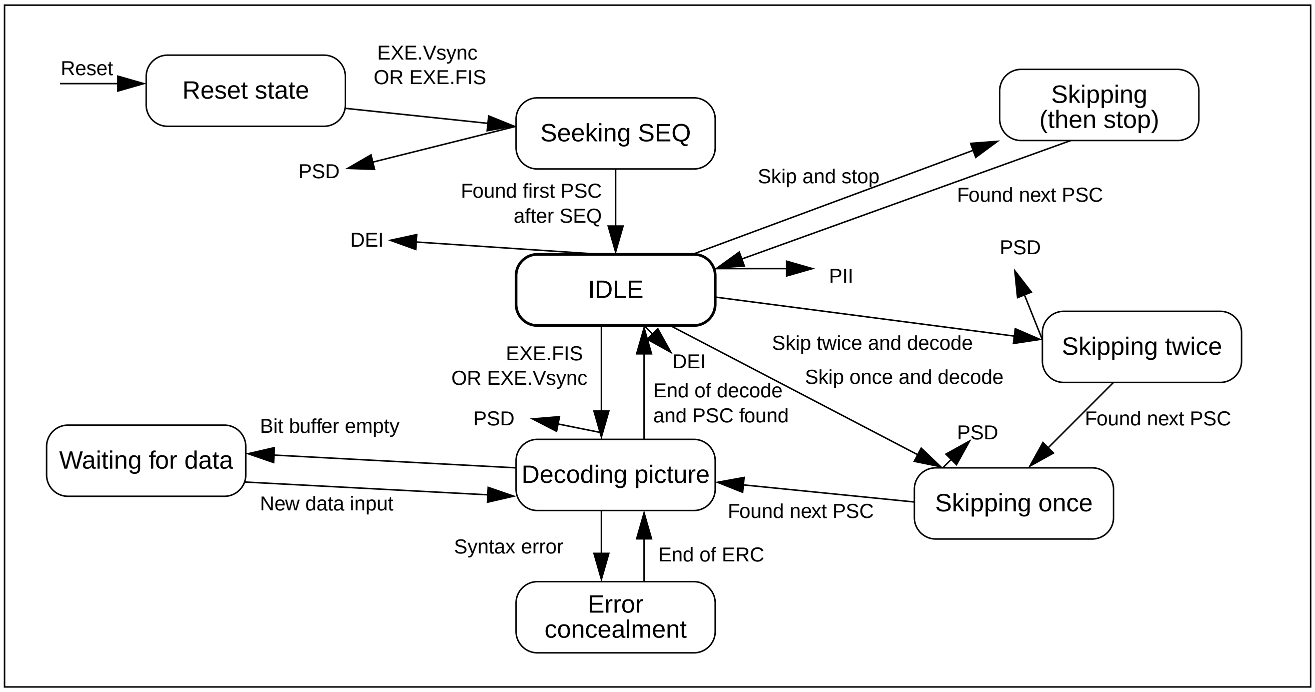


Figure 18.10 Task control state diagram

The instruction bits which affect state transitions are **VID_TIS.EXE** and **VID_TIS.FIS**. The events to which the controller responds are:

- VSYNC, which could be a *VSYNC top* or a *VSYNC bottom* and
- *IDLE* representing the idle state of the pipeline.

### 18.8.2 Error recovery and missing macroblock concealment

There are four levels of error detection and recovery available in the video decoder:

- bit-stream syntax error detection with the option of automatic missing macroblock concealment;
- bit-stream semantic error detection with the option of automatic concealment or skip to the next picture;
- pipeline overflow or underflow error detection;
- user-initiated skip to next sequence using soft reset.

#### Syntax error detection and concealment

In normal operation of the STi5500, error concealment must always be enabled, i.e. **VID_CTL.DEC** should be reset.

If the VLD detects a syntax error in the bit-stream, the pipeline will copy macroblocks from the previous picture using the motion vectors reconstructed for the previous row of macroblocks in the current picture, while scanning the bit-stream until a slice start code is detected. At this point normal decoding resumes. If the slice in which the error occurred was the last one in the picture, concealment will continue until the end of the picture, at which time the pipeline stops normally (assuming that the following picture start code is intact).

Concealment of macroblocks is carried out by using the vectors of the macroblock immediately above the lost macroblock. The pipeline is able to store one row of such information, for a decoded picture size of up to a maximum of 46 macroblocks. Two vectors are stored for each macroblock in the row.

The concealment macroblocks are accessed using the pointers **VID_FFP** and **VID_BFP**. Lost macroblocks in the first row are copied directly from the previous pictures (i.e. as P-macroblocks with zero motion vectors). If an intra picture is coded with concealment motion vectors, these will be used. If not, then the concealment will be a simple copy from the previous picture using zero vectors. Even in intra pictures, the pointer **VID_FFP** must be set up.

Table 18.2 shows the rules that are used for fetching concealment macroblocks.

| Picture type | Macroblock type | Fetch rule |
|---|---|---|
| I-picture | I-macroblock without vectors | Copy with zero motion. |
| | I-macroblock with vectors | Copy as forward predicted macroblock. |
| P-picture | I-macroblock without vectors | Copy with zero motion. |
| | I-macroblock with vectors | Copy as forward predicted macroblock. |
| | P-macroblock | Copy using stored vector. |
| | P-field-macroblock | Copy in field mode using both vectors. |
| | Skipped macroblock | Copy with zero vector. |
| | Dual-prime macroblock | Copy using stored vector. |
| B-picture | I-macroblock without vectors | Copy with zero motion. |
| | I-macroblock with vectors | Copy as forward predicted macroblock. |
| | Forward macroblock | Copy using stored vector. |
| | Backward macroblock | Copy using stored backward vector. |
| | Bidirectional macroblock | Only the forward vectors are stored, concealed as forward macroblock. |
| | Skipped macroblock | Copy in frame mode using the same mode and vectors as the previous macroblock. |

Table 18.2 Rules for fetching concealment macroblocks

If an error is detected in the bit-stream before it enters the parser, then an error start code can be inserted into the bit-stream in order to initiate concealment. However, when doing this there are certain restrictions on the placement of the error start code in order to avoid emulation of other start codes. An Application Note is available on this topic.

### Overflow or underflow error

An overflow error occurs whenever the pipeline reconstructs more macroblocks than are defined by the decoded picture size, **VID_DFS**. This can occur when the input data to the decoder contains undetected errors. This condition is signalled by bit **VID_STA.SER**. Decoding is automatically halted when this error is detected. In order to restart decoding a pipeline reset must be performed.

An underflow error occurs whenever the pipeline reconstructs less macroblocks than are defined by the decoded picture size, **VID_DFS**. This condition is signalled by bit **VID_STA.PDE**. Decoding is automatically halted when this error occurs. In order to restart decoding a pipeline reset must be performed.

**Soft reset**

The effect of this last resort action is described in section 18.2.

## 18.9 PES Parser

### 18.9.1 Overview

The block is situated between the MCU interface and the compressed data FIFOs of the video/ audio core.

- Bit rate: 100 Mbits/sec (maximum burst).
- Input streams allowed:
  - MPEG-2 packetized PES, to ISO 13818-1;
  - MPEG-1 system layer, to ISO 11172-1.

The STi5500 accepts PES streams in the same way as pure audio or video streams are accepted. The interface remains unchanged; a common data and address bus with separate request and data strobes for compressed audio and video data. In the case of packetized elementary stream data demultiplexed from an MPEG-2 transport stream, the data stream consists of concatenated, incomplete packets of audio and video PES. To handle this configuration the STi5500 contains two separate parsers one for the audio and one for the video data. Each parser is activated by one of the compressed data strobe signals. As the audio or video data is input it is demultiplexed by each parser and the audio/video streams placed in their respective buffers.

In the case of program stream data or MPEG-1 systems stream data the audio and video packets are complete so that a single parser (and compressed data strobe) can be used the packets being internally separated into video and audio streams. If desired the two parsers can still be used but the packets must be separated outside the STi5500. For more details refer to Figure 18.11.

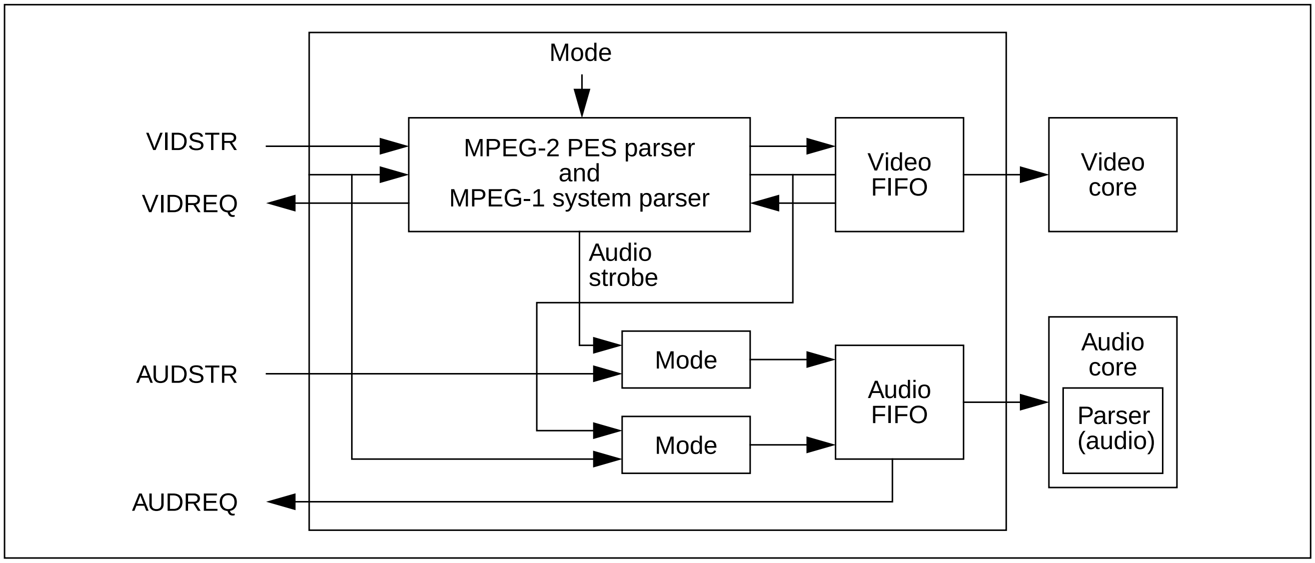


Figure 18.11 System parser internal architecture

When the device is configured to accept PES, the audio/video strobe and request signals will refer to packetized audio/video PES streams. The parser will extract audio and video bit-streams in accordance with the programmed stream ID contained in **PES_CF1** for the audio stream and **PES_CF2** for the video stream. Any audio or video packets which are not selected for decode (because their stream IDs do not match the programmed values) are discarded. When used for decoding program streams or MPEG-1 system streams a single strobe can be used to input all data. The audio, video and system level data are automatically separated internally to the decoder. Support is provided for time stamp association by the decoder.

Decode or display time stamps (DTSs or PTSs, selected by **PES_CF1.SDT**) are stored in an internal FIFO during parsing. When the image corresponding to these time stamps is decoded (or about to be decoded in the case of video) the corresponding time stamp is made available and a flag or interrupt is given. A global view of the parser and ancillary blocks is shown in Figure 18.12.

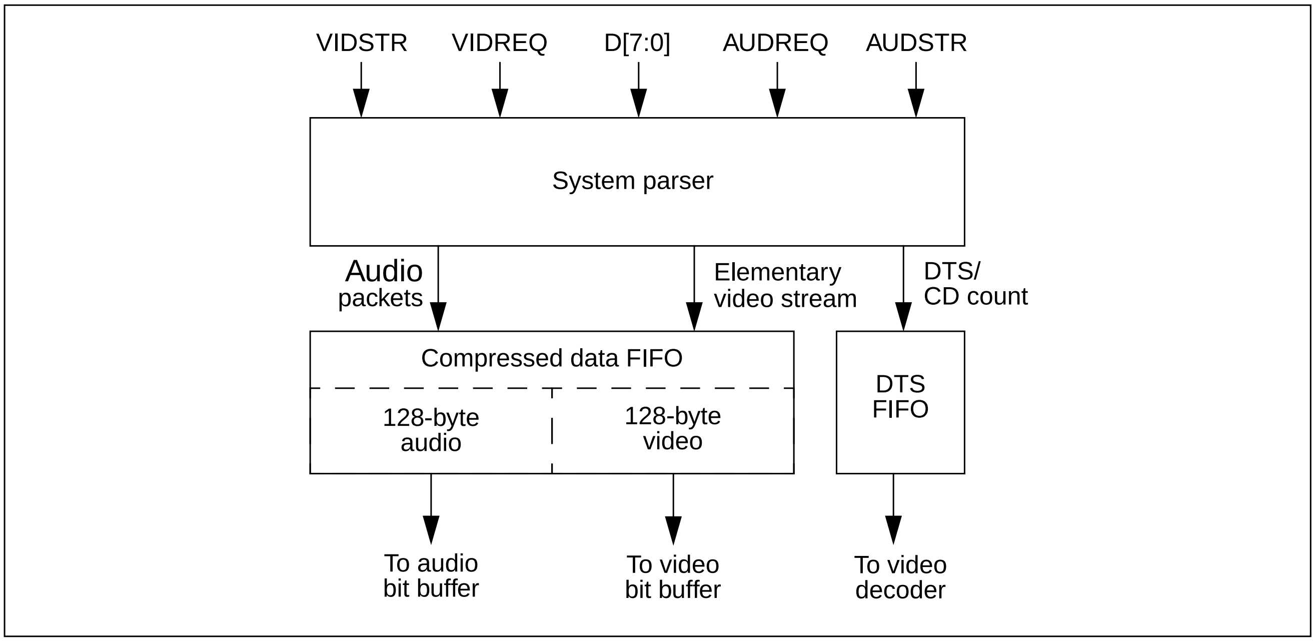


Figure 18.12 PES parser block diagram

### 18.9.2 Functional modes

The parsers are enabled by setting **PES_CF2.SS** for the video parser and **AUD_ISS[2:0]** for the audio parser. Depending on the required mode one or both of the parsers will be required. When either of these registers is reset the subsequent decoder will accept either pure audio or video streams.

Four different modes can be configured with the two mode bits contained in **PES_CF2[7:6]**:

- Mode 0: Automatic configuration.The parser will examine the incoming stream and self-configure for decode. The mode selected can be read back from **PES_TM2[1]**.
- Mode 1: MPEG-1 system stream decode. Single data strobe input format.
- Mode 2: MPEG-2 PES decode. Twin data strobe input format. This corresponds to the most common mode of entry of data into the circuit.

- Mode 3: MPEG-2 whole PES audio/video packets. Single data strobe input format. This can be used to decode MPEG-2 program streams.

These modes are summarized in Table 18.3 and Table 18.4.

For modes using a single data input strobe, $\overline{\textbf{VIDSTR}}$ is used.

| **Mode Required** | **PES_CF2_MOD** | **AUD_ISS [2:0]** | **Data Strobes** |
|---|---|---|---|
| Automatic Mode[1] | 00 | 101 | 1 or 2 |
| MPEG-1 System | 01 | 001 | 1 |
| MPEG-2 PES | 10 | 100 | 2 |
| Program Stream | 11 | 100 | 1 |

Table 18.3 PES Modes

Note 1. This mode only works with MPEG-1 and MPEG-2-PES; it cannot be used with packetized PES.

| **Strobes** | $\overline{\textbf{AUDSTR}}$ | $\overline{\textbf{VIDSTR}}$ |
|---|---|---|
| **Bits SS/MOD** | | |
| **PES_CF2.SS** = 0 | Audio Elementary stream<br>Packet MPEG1 PES | Video Elementary Stream |
| **PES_CF2.SS** = 1<br>**PES_CF2.MOD** = 00 | Depending on detection see MOD = 01 or MOD = 10 | Automatic Mode (MOD = 01 or MOD = 10) |
| **PES_CF2.SS** = 1<br>**PES_CF2.MOD** = 01 | Not used | MPEG1 System Stream |
| **PES_CF2.SS** = 1<br>**PES_CF2.MOD** = 10 | Audio Elementary stream<br>Packet MPEG1 PES | Program Stream PES |
| **PES_CF2.SS** = 1<br>**PES_CF2.MOD** = 11 | Not used | Program Stream |

Table 18.4 PES Strobes

# 19 Sub-picture decoder

## 19.1 Introduction

A hardware sub-picture decoder is integrated in the STi5500. The sub-picture bit-buffer that contains sub-picture units (SPU) is integrated in SDRAM external memory and has a programmable size. Its position and size can be randomly chosen in multiples of 2 Kbytes. The sub-picture bit buffer is set up at power up reset. During player operation, its size and location are constant.

Compressed data is input into the bit-buffer using a DMA or by a CPU write. Once control is given to the sub-picture decoder it is autonomous until stopped by software control. The sub-picture decoder can decode complete sub-picture units consisting of a sub-picture unit header, compressed pixel data and the display control sequence table without any interaction from the ST20.

Figure 19.1 shows the architecture of the sub-picture decoder.

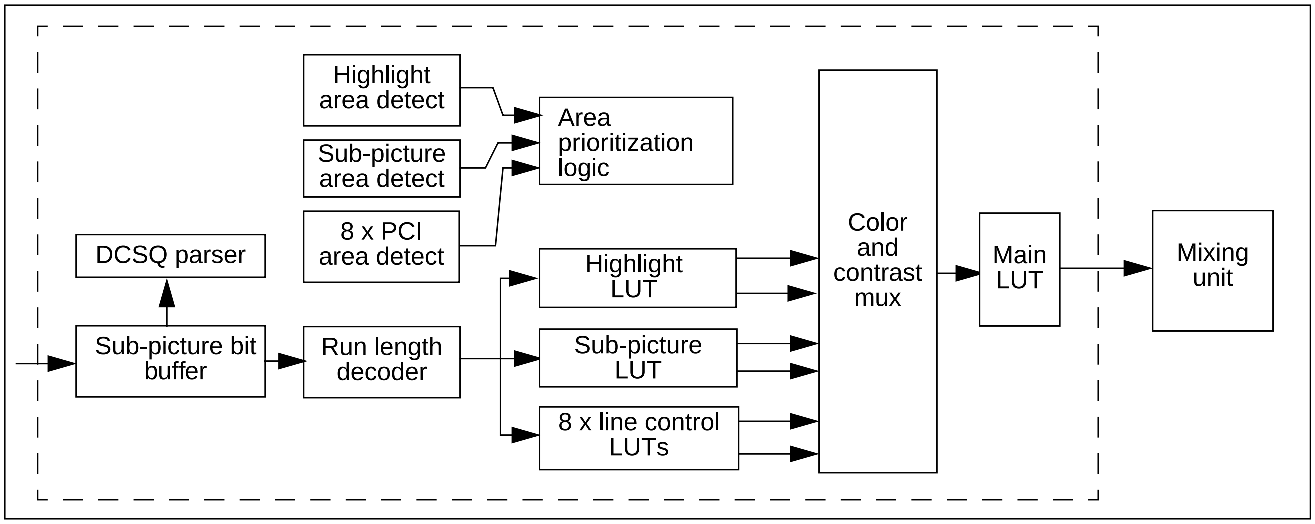


Figure 19.1 Sub-picture unit architecture

## 19.2 Buffer management and pointers

There are four programmable registers to control the sub-picture bit buffer read and write processes, as shown in Figure 19.2:

- Bit buffer base address (**VID_SPB**). This is an offset relative to the ST20 SDRAM base address. It is programmed in units of 2 Kbytes.
- Bit buffer end address (**VID_SPE**). This address is an offset relative to the ST20 SDRAM base address. It is programmed in units of 2 Kbytes.
- Bit buffer read pointer (**VID_SPRead**). It is set by software for each sub-picture unit. This is done before control is given to the sub-picture hardware decoder. This register is double buffered. The shadow register is updated with each field VSYNC event. This pointer is an offset relative to the ST20 SDRAM base address. It is programmed in units of 64-bit words (see Figure 19.1).

- Bit buffer write pointer (**VID_SPWrite**). It is set by the ST20 before transferring each sub-picture unit into the bit buffer. This pointer is an offset relative to the ST20 SDRAM base address. It is programmed in units of 64 bit words.

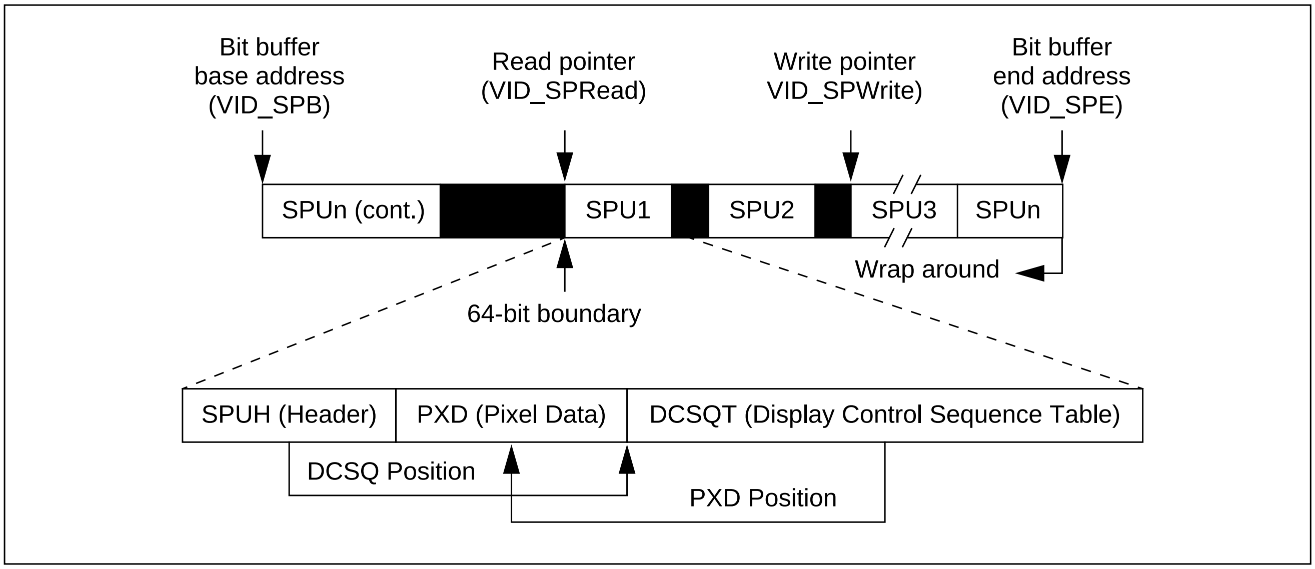

Bit buffer base address (VID_SPB)
Read pointer (VID_SPRead)
Write pointer VID_SPWrite)
Bit buffer end address (VID_SPE)
SPUn (cont.)
SPU1
SPU2
SPU3
SPUn
Wrap around
64-bit boundary
SPUH (Header)
PXD (Pixel Data)
DCSQT (Display Control Sequence Table)
DCSQ Position
PXD Position

Figure 19.2 Buffer management

## 19.3 Operation

Each sub-picture unit data buffer start position is programmed using the register **VID_SPWrite**. Subsequently the sub-picture header, the pixel data, the display control sequences are sent via fifos to the sub-picture decoder. Write into fifos is done by DMA or by CPU write. Only data belonging to the sub-picture unit (SPUH, PXD, DCSQT) are transferred int the sub-picture bit buffer. Sub-picture pack headers are removed by the software demultiplexor.

The decoder reads the header of the first packet (see Figure 19.3) and jumps to the first display control sequence using the command pointer.

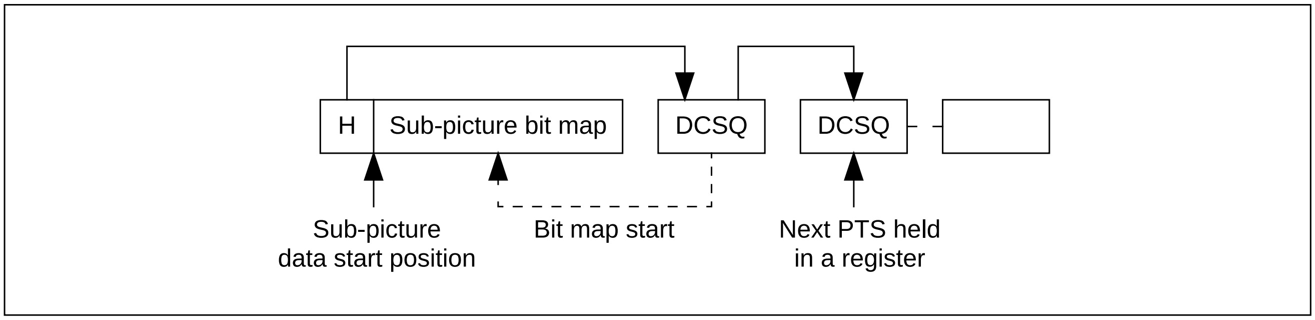


Figure 19.3 Sub-picture unit structure

The instructions found in the DCSQ packets enable the sub-picture unit to program the palettes, set mixing factors etc. for each region. The DCSQ packets also contain a time stamp which indicates to which image the sub-picture information refers.

This information is related to a local time for this sub-picture unit. The micro should enable a given sub-picture unit at the right global time via some registers: data buffer start position, start sub-picture unit status bit.

The overall control of the sub-picture decoder is performed by software.

The final information in the DCSQ packet is the region size (rectangle) and the relative position, in bytes, of the bit-map start.

A key point here is that the sub-picture decoder must read beyond the end of the DCSQ packet in order to verify the next PTS. With this information held in a register, the sub-picture decoder knows, in advance, when to change the DCSQ or bit-map information. The sub-picture unit simply executes the same DCSQ until the image corresponding to the next time-stamp is reached.

This is done at the beginning of every field so that the sub-picture decoder can load all the relevant information from DCSQ before the first sub-picture pixel is required.

The sub-picture region declaration is held in registers in the decoder so that the sub-picture decoder is turned on and off at the correct position on the screen (refer to Figure 19.4). The bit-map start pointer indicates where, in the bit map data, to start decoding. When the correct image, corresponding to the local time stamp contained in the DCSQ, should be displayed the sub-picture controller enables the sub-picture decode for that image.

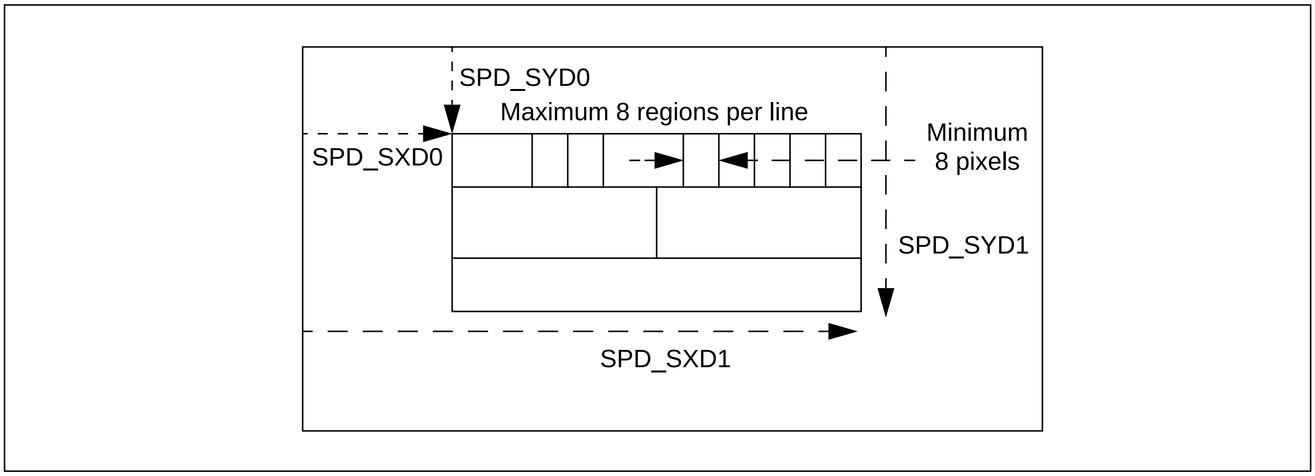


Figure 19.4 Sub-picture region declaration

A pause mode is defined in the sub-picture decoder. As explained previously, the sub-picture decoder is autonomous within a sub-picture unit.

This means that the DCSQ switching is timed automatically using an internal 90kHz clock. During video trick modes, where the video stream may be frozen or slowed down the same thing should be possible with the sub-picture decoder in order to maintain the synchronization between the two streams.

A pause mode is implemented for the sub-picture decoder which stops the 90kHz counter and therefore pauses the sub-picture decoder. This is controlled using the **P** field in the **SPD_CTL1** register and is synchronized to the VSYNC signal. This control bit can therefore be used as a pause and a single step control bit.

The sub-picture decoder registers are put together in the sub-picture memory map except:

- sub-picture software reset (bit **VID_CTL.SPR**),
- sub-picture pause mode (**VID_DCF.SPP** bit),
- sub-picture FIFO full (bit 18 of **VID_ITS** and **VID_STA** register).

## 19.4 Sub-picture display

### 19.4.1 Look-up tables

There are 11 look-up tables inside the sub-picture decoder:

- 1 highlight LUT (2 bits to 4 bits mapping)
- 1 sub-picture LUT (2 bits to 4 bits mapping)
- 8 PCI LUTs (2 bits to 4 bits mapping)
- 1 main LUT (4 bits to 24 bits mapping)

The sub-picture and PCI LUTs are automatically supplied by the decoder itself (sub-picture commands contained in the SPU). The highlight and main LUTs need to be loaded by the ST20 (**SPD_HCN**, **SPD_HCOL**, **SPD_LUT** registers).

The output of the sub-picture main LUT is mixed with the MPEG video. The contrast value between these two sources is set by the SET_CONTR DCSQ command, by the PCINFs of a CHG_COLCON command or by a highlight color information (the highlight LUT has the highest priority, followed by the PCI LUTs. The sub-picture LUT has the lowest priority).

The mixed video is a 24 bits Y, $C_r$, $C_b$ video where:

$$Y_{MIXED} = [Y_{MPEG} \text{ x } (16 - k) + Y_{SUBP} \text{ x } k] / 16$$

$$Cr_{MIXED} = [Cr_{MPEG} \text{ x } (16 - k) + Cr_{SUBP} \text{ x } k] / 16$$

$$Cb_{MIXED} = [Cb_{MPEG} \text{ x } (16 - k) + Cb_{SUBP} \text{ x } k] / 16$$

k = 0 if contrast value from high light, sub-picture, PCI LUTs = 0

k = contrast value + 1 if contrast value > 0

### 19.4.2 Sub-picture areas

The active sub-picture decoding area can be 720 x 576 or 720 x 480 pixels. In order to align the sub-picture decoding area with the video decoding area, the upper left corner of the active sub-picture decoding area has to be set by software, using the registers **SPD_XD0** and **SPD_YD0**. The same semantics have been defined as for the video decoder, as shown in Figure 19.5. The active sub-picture display area is defined in a similar manner, using the **SPD_SXD0**, **SPD_SYD0**, **SPD_SXD1** and **SPD_SYD1** registers.

The highlight area is defined through **SPD_HLSX**, **SPD_HLSY**, **SPD_HLEX**, **SPD_HLEY** registers and is set by software.

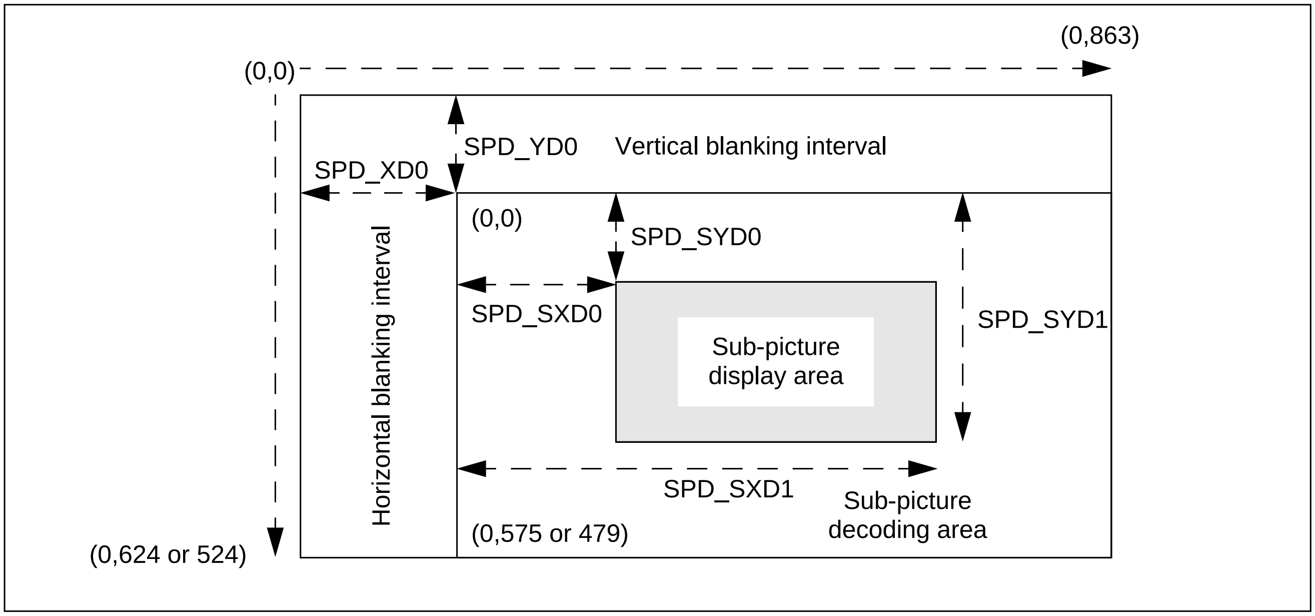


Figure 19.5 Sub-picture areas

# 20 MPEG audio decoder with AC-3 interface

The audio decoder receives compressed data from an audio bit-buffer which is integrated in the external SDRAM.

When the external AC-3 interface is used, the compressed/PCM data also goes through the audio bit-buffer. The audio bit-buffer is memory mapped into the register/compressed data address space in the same way as the video and sub-picture decoders. Data is transferred either using a compressed data DMA engine or CPU writes.

The audio decoder is completely autonomous, needing no interaction from software during decoding, apart from exceptions such as error conditions and ancillary data in the audio stream.

## 20.1 PCM output

### 20.1.1 Interface and output formats

The decoded audio data is output in serial PCM format.

The interface consists of the following signals:

- PCMDATA - PCM serial data output
- SCLK - PCM clock output
- LRCLK - Left/right channel select output
- PCMCLK - PCM clock input.

Output precision is selectable to be either 16 bits/word or 18 bits/word by setting the output precision select register, **AUD_P18**. In 16-bit mode, data may be output either with the most significant bit first or least significant bit first, selected by the output order select register, **AUD_ORD**. When 18-bit data is selected, 32 bits are output for each channel. The data-in-front register, **AUD_DIF**, is used to position the 18 data bits either at the beginning or at the end of each 32-bit frame. The **AUD_FOR** register is used to select standard or $I^2S$-compatible format when 18-bit precision is selected.

Figure 20.1 shows the five different output formats which are possible. **AUD_ORD** only has significance in 16-bit mode, while **AUD_DIF** only has significance in 18-bit mode. **AUD_FOR** only has significance in 18-bit mode and when **AUD_DIF** = 1. The last option shown in Figure 20.1 is compatible with the $I^2S$ format.

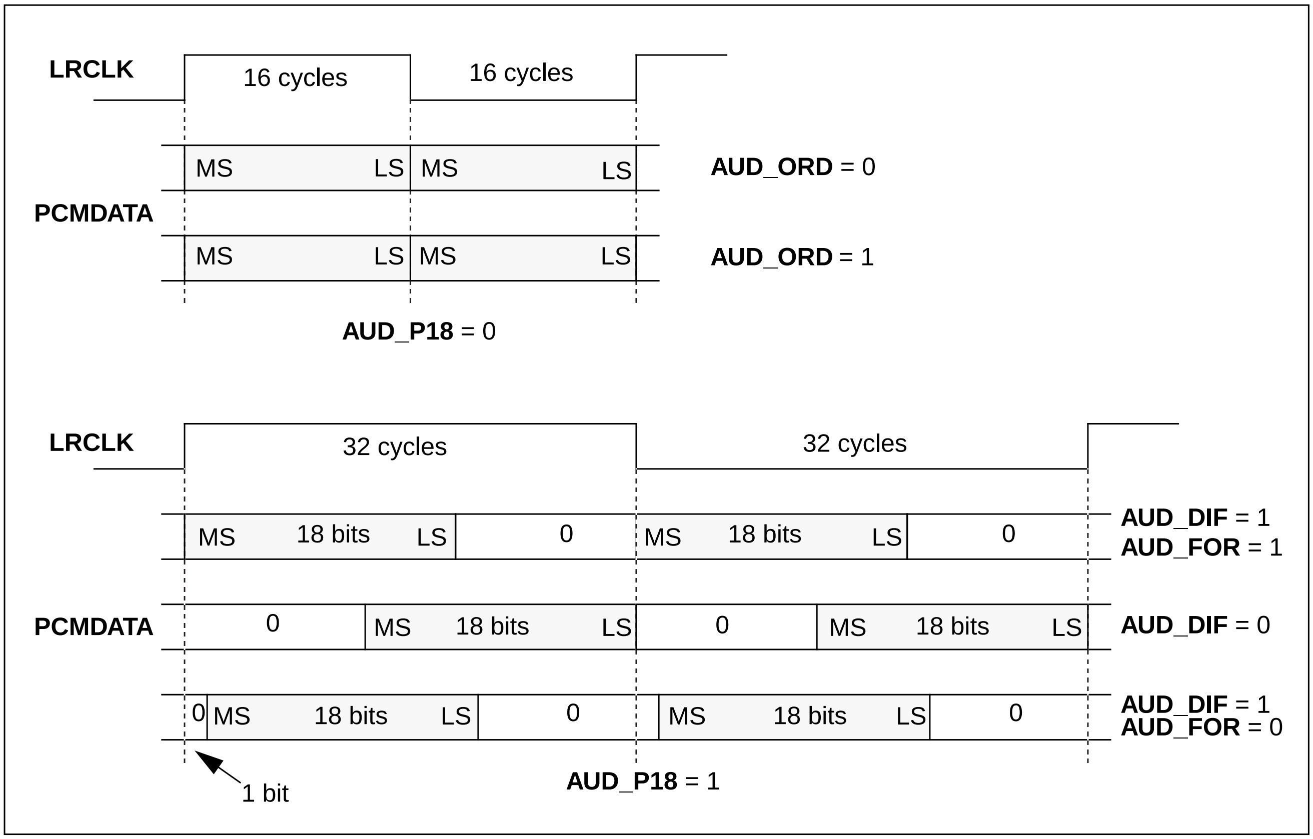


Figure 20.1 PCM output formats

The polarity of the PCM serial output clock, SCLK, and the left/right channel selection, LRCLK, are selected by the 1-bit registers **AUD_SCP** and **AUD_LRP** respectively.

Figure 20.2 shows the two polarities of SCLK. Normally, the DAC will sample LRCLK and PCMDATA on the rising edge of SCLK in the first case, and on the falling edge of SCLK in the second. The first option (**AUD_SCP** = 0) is the one normally used in I$^2$S systems.

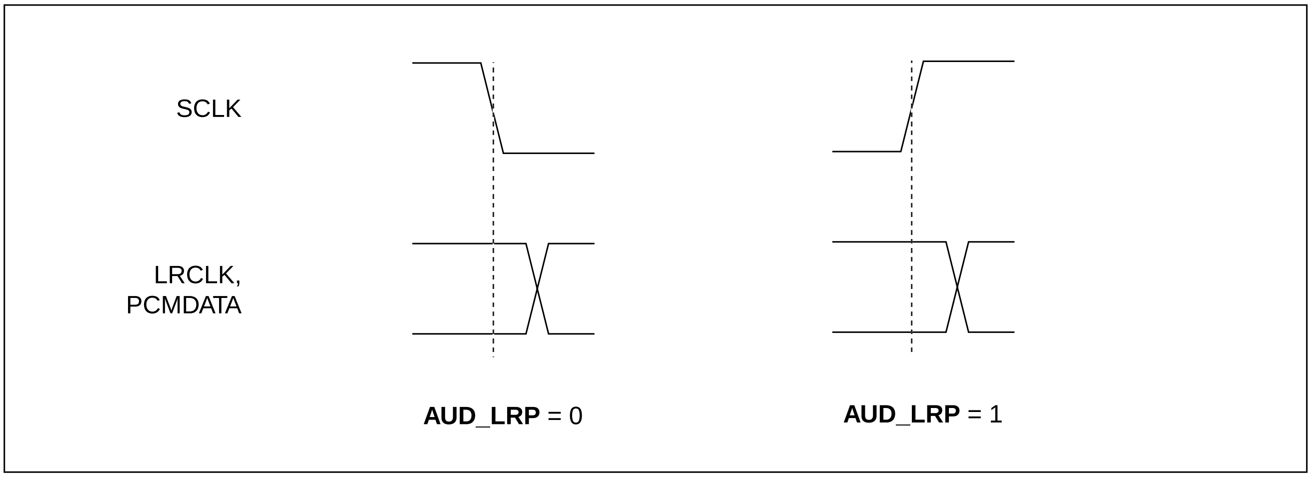


Figure 20.2 SCLK polarity

Figure 20.3 shows how the polarity of LRCLK is selected. The second option (**AUD_LRP** = 1) is compatible with the I$^2$S format.

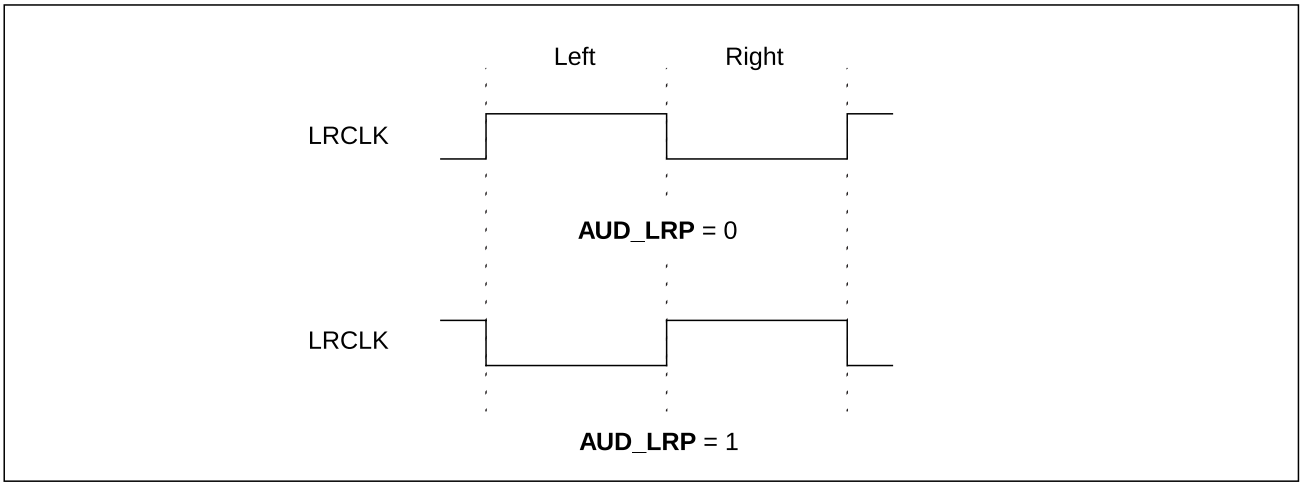


Figure 20.3 LRCLK polarity

### 20.1.2 PCM clock generation

The PCM serial clock SCLK is derived from the clock input PCMCLK. The frequency of PCMCLK may be equal to the PCM output bit rate or it may be an integer multiple of this, allowing the use of oversampling D-A converters. In many applications PCMCLK is externally synchronized to the compressed audio bit stream.

SCLK is derived by dividing PCMCLK by the contents of the divider register, **AUD_DIV**. This number, in the range 0 to 63, defines the ratio of the frequency of the PCM bit clock SCLK, to that of PCMCLK, according to the relationship:

$$f_{SCLK} = \frac{f_{PCMCLK}}{2 \times (\mathbf{AUD\_DIV} + 1)}$$

For example, **AUD_DIV** is loaded with 0, the frequency of SCLK is one half of the frequency of PCMCLK, while if **AUD_DIV** is loaded with 63, the frequency of SCLK is 1/128th of the frequency of PCMCLK.

The value of **AUD_DIV** = 16 is reserved. If this number is loaded, the divider is bypassed and the frequency of SCLK is equal to the frequency of PCMCLK.

**AUD_DIV** must be set up before the output of SCLK starts. This can be done by first disabling the PCM outputs by de-asserting the MUTE and PLAY commands and then writing to the **AUD_DIV** register. Once the register is set up, the MUTE or PLAY or both commands can be asserted. **AUD_DIV** cannot be changed “on the fly”.

The frequency of LRCLK is given by:

$$f_{LRCLK} = \frac{f_{SCLK}}{32} \qquad \text{for 16-bit PCM output}$$

$$f_{LRCLK} = \frac{f_{SCLK}}{64} \qquad \text{for 18-bit PCM output}$$

### 20.1.3 Interrupts associated with PCM output

There are two interrupts associated with the PCM output. These are:

- interrupt 8, indicating PCM buffer underflow. This is generated (unless masked) when a new output sample is required and the PCM buffer is empty. The PCM buffer, which contains up to 64 samples (i.e. 64 word-pairs in stereo), receives the decoded outputs from the DSP core. If the buffer is empty the output sample will have the value zero, but decoding will not stop. If the PCM buffer becomes full, decoding will stop, but PCM output will not be affected.
- interrupt 14, indicating output of a new frame. This is generated (unless masked) whenever the first bit of a frame appears at the PCM output.

## 20.2 Audio decoder control

### 20.2.1 Play and mute

Once initialized and configured, decoding and output of PCM data is controlled by the commands PLAY and MUTE.

The command PLAY is asserted when the **AUD_PLY** register is written to. The command MUTE is as The actions of the PLAY and MUTE commands are specified in Table 20.1.

| MUTE | PLAY | Function |
|---|---|---|
| de-asserted | de-asserted | No output or decoding.<br>SCLOCK, LRCLK, PCMDATA all move into their inactive state. LRCLK completes its current cycle and stops, SCLK completes its last cycle in the second LRCLK frame and stops.<br>Decoding stops when all internal buffers become full. |
| de-asserted | asserted | Normal decoding and PCM output.<br>When PLAY is re-asserted, PCMDATA resumes where it left off, without data loss. |
| asserted | de-asserted | PCM clocks only, no decoding.<br>PCMDATA becomes low after the output of the last complete sample. LRCLK and SCLK are not stopped.<br>Decoding stops when all internal buffers become full.<br>When PLAY is re-asserted, PCMDATA resumes where it left off, without data loss. |
| asserted | asserted | Decoding and muted output (soft mute).<br>PCMDATA gradually decays to zero.<br>Decoding continues normally. Data consumed as if output were playing. |

Table 20.1 Mute and play functions

### 20.2.2 Restart

The restart procedure is invoked when it is required to flush all buffers and restart decoding immediately.

Restart is initiated by writing 0 or 1 to the **AUD_RST** register, after which it is automatically restored to the 0 state. A restart initiates the following actions:

- The **AUD_ITR** and **AUD_ITN** registers are cleared.

- All data buffers are cleared.
- The **AUD_MUT**, **AUD_PLY** and all others registers (except those mentioned above) remain in their existing state.
- Register access is not disabled. However, compressed data input may be interrupted
- The **AUD_RST** register is cleared; compressed data input can restart.

### 20.2.3 Bit stream synchronization

The compressed input bit stream must be synchronized before the decompression step may begin. This is done by looking for synchronization words inserted into the data stream at encoding. Synchronization must be done both at the audio frame and at the system packet layer if present.

At the packet level, the audio decoder will look for a valid start code, doing a bit by bit search. Once an audio packet is found, the decoder extracts the presentation time stamp (PTS) if present and starts the audio synchronization described below.

At the audio frame level, there is a non-unique sync word at the beginning of the header. The STi5500 attempts to find this sync word by doing a bit by bit search. When found the action taken depends on the contents of the **AUD_LCK** and **AUD_LAT** registers.

### 20.2.4 Packet Level Synchronization

The complete algorithm is given in Figure 20.4.

To help the synchronization algorithm ignore an emulated packet synchronization word, it is possible to extend the packet start code to be matched. Depending to the content of registers **AUD_SYN**, **AUD_SID** and **AUD_IDE**, synchronization can be made on the 24-bit *packet_start_code_prefix* or can be extended to the *stream_id* field.

Synchronization mode depends on the type of packets received by the STi5500. The decoder can receive either:

1 Multiplexed audio/video bitstream (**AUD_SYN** = 0)

In this case the STi5500 can receive both video and audio streams multiplexed together. Packet synchronization is possible only on the 24-bit start code.

All packets are used by the synchronization algorithm but all non-audio packets and, if **AUD_IDE** is set, all audio packets which have a *stream_id* which does not match the **AUD_SID** register value, are not decoded.

2 Multiplexed audio bitstream (**AUD_SYN** = 1)

In this case, the STi5500 expects to receive only multiplexed audio streams. Synchronization is performed on 27 bits (24 bits *packet_start_code_prefix* + 3 first bits of *stream_id*).

All packets are used by the synchronization algorithm but if **AUD_IDE** is set, all audio packets that have a *stream_id* which does not match the **AUD_SID** register value are not decoded.

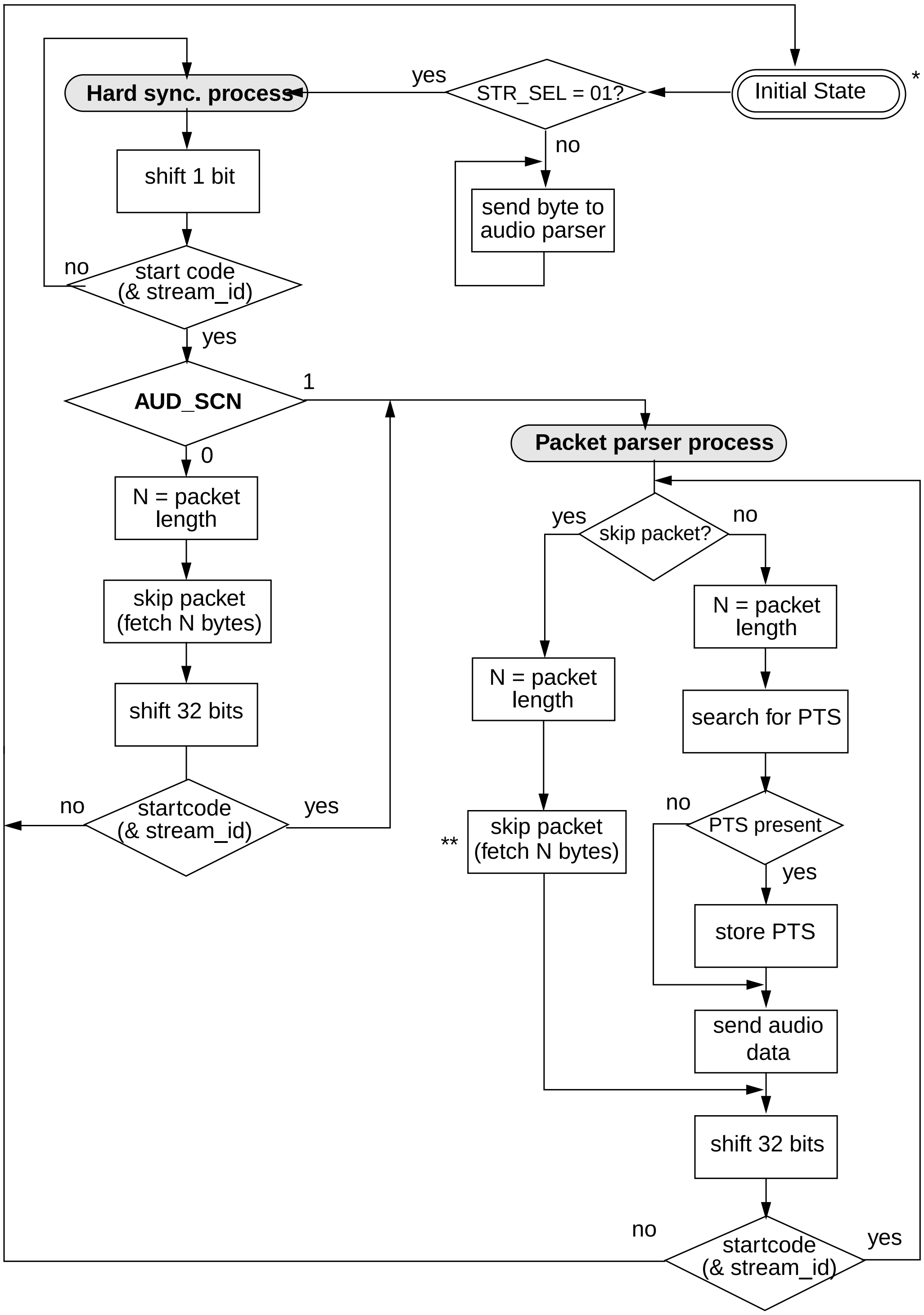
3 Single audio bitstream (**AUD_SYN** = 2, **AUD_IDE** =1)

Synchronization is performed on 32 bits.

All packets are used by the synchronization algorithm, and all audio packets that have a *stream_id* which matches the **AUD_SID** register value are decoded.

The **AUD_SCN** register is also taken into account in the global synchronization algorithm.

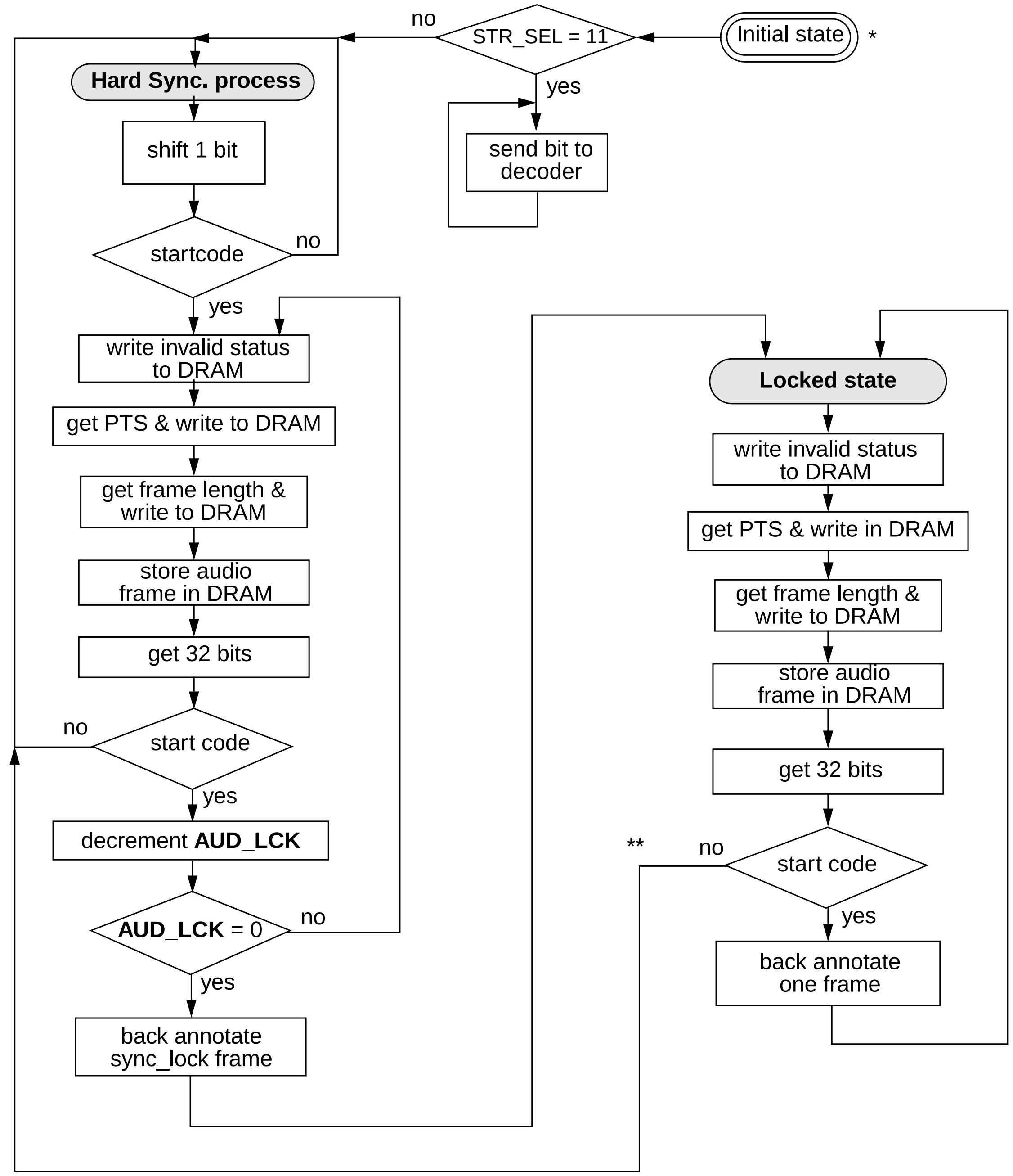
If **AUD_SCN** = 1, after the first packet synchronization word is found the STi5500 is considered to be synchronized. If **AUD_SCN** = 0, after the first packet synchronization word is found, the STi5500 must read the packet length and confirm synchronization by finding the next synchronization word in the correct position.



* on reset, packet parser goes to “Initial State”

** skip video or audio packet that does not have valid stream_id

Figure 20.4 Packet synchronization algorithm



* on reset audio parser goes to initial state

** on loss of packet sync audio parser goes to “Hard Sync” process

Figure 20.5 Audio frame synchronization algorithm

### 20.2.5 Audio frame synchronization

The synchronization algorithm is given in Figure 20.5.

As the audio *syncword* can be emulated in the bit stream, it is useful to extend this audio start code to avoid the detection of a false sync word. Each time the STi5500 detects a false sync word during the synchronization process, the delay to reach the locked state increases.

The **AUD_SYE** register is used for this purpose. When no field of the **AUD_SYE** register is enabled, the STi5500 saves the layer and sampling frequency information after synchronization is achieved. This aids the task of resynchronization, should synchronization be lost owing to an error in the audio data or the system layer. This internal register is disabled on **AUD_RST** or RESTART and will not be reinitialized until the audio parser is synchronized.

The **AUD_LCK** register specifies how many valid synchronization words after the initial one have to be found before entering the locked state. The highest value of **AUD_LCK** (i.e. 3) is assumed when the **AUD_SYE** register has its default value. The definition of a valid synchronization word depends on the **AUD_LAT** register value.

A valid synchronization word is a sequence of bits matching the expected word.

In free-format mode one additional register (**AUD_FFL**) can be used. The **AUD_FFL** register is a way of specifying the length of an audio frame in free-format mode. This register is 16 bits long and contains the length of the frame in bits.

### 20.2.6 Error recovery and concealment

The STi5500 audio decoder is able to recover from certain detectable errors. For this purpose it has a number of user-selectable error concealment modes.

Detectable errors may be caused by a bad audio frame CRC or by loss of synchronization. Concealment is similar, but may be selected independently by setting the **AUD_CRC** and **AUD_SEM** registers.

The register **AUD_CRC** defines the action which will be taken upon detection of a CRC error in an input frame.

| Value | Meaning |
|---|---|
| 00 | Disable CRC detection and error concealment |
| 01 | Mute on detection of CRC error |
| 10 | Illegal |
| 11 | Skip invalid frame |

Table 20.2 **AUD_CRC** register coding

The register **AUD_SEM** defines the action which will be taken upon detection of a synchronization error, using the coding in Table 20.3.

| Value | Meaning |
|---|---|
| 00 | Ignore error |
| 01 | Mute on detection of synchronization error |
| 10 | Illegal |
| 11 | Skip invalid frame |

Table 20.3  **AUD_SEM** register coding

### 20.2.7 Ancillary data extraction

The ancillary data which may be held at the end of audio frames can be extracted and read from the **AUD_ANC** register. This register constitutes a 32-bit FIFO. The first bit of ancillary data received is stored in bit **AUD_ANC[0]**.

The extraction of ancillary data in **AUD_ANC** is started by enabling interrupt 7. An interrupt 7 is generated when either:

1. 32 bits of ancillary data have been received from the bit stream and written into **AUD_ANC**, i.e. when it is full, or
2. the end of a frame is reached.

Register **AUD_ADA** holds the number of bits available in the ancillary data buffer, **AUD_ANC[31:0]**.

When **AUD_ANC[31:24]** is read, interrupt 7 is cleared, **ANC_ADA** is cleared and the ancillary data buffer is reinitialized.

Decoding stops if the STi5500 tries to write data into **AUD_ANC** when it is full. The normal response would be to read **AUD_ADA** and then **AUD_ANC**. However, if interrupt 7 is disabled (by resetting bit **AUD_ITN[7]**), decoding will continue and the registers **AUD_ANC** and **AUD_ADA** will retain their contents until **AUD_ANC[31:24]** is read.

If **AUD_ANC** is not read at the end of the frame, and it is not full, ancillary data bits in the next frame will be appended.

## 20.3 AC-3 interface

The interface to the off-chip AC-3 Audio Decoder is composed of two serial buses The $I^2C$ bus is used for control and a synchronous serial interface for compressed data transfer.

### 20.3.1 Input / output description

The external AC-3 interface uses six signals. Four of these signals are multiplexed with the internal MPEG-1 decoder output signals. The signals are described in Table 20.4.

| Signal Name/AC-3 | Signal Name MPEG-1 | Type | Description |
|---|---|---|---|
| A_C_DATA | PCM_DATA | Out | AC-3 packet data or PCM serial data |
| A_C_STB | PCM_CLKOUT | Out | AC-3 packet strobe or PCM clock |
| A_C_REQ |  | In | AC-3 data request |
| A_WORD_CLK | LRCLK | Out | AC-3 word clock or PCM L/R clock |
| A_PTS_STB |  | In | AC-3 PTS strobe |
| A_IRQ |  | In | AC-3 Interrupt request |
|  | PCM_CLKIN | In | PCM clock input from VCXO |

Table 20.4 External AC-3 decoder interface

A schematic of the interface along with timing is shown in Figure 20.6.

**A_C_REQ** is active when the AC-3 decoder is capable of accepting data and **A_C_STB** is used to strobe the data into the audio decoder on the rising edge. The signal **A_WORD_CLK** is the **A_C_STB** signal divided by 32. It is phased so that the transition coincides with a byte boundary. This signal can be used as a framing signal for certain AC-3 decoders.

The **A_PTS_STB** is used to latch the value of a free running timer in the STi5500 for clock recovery.

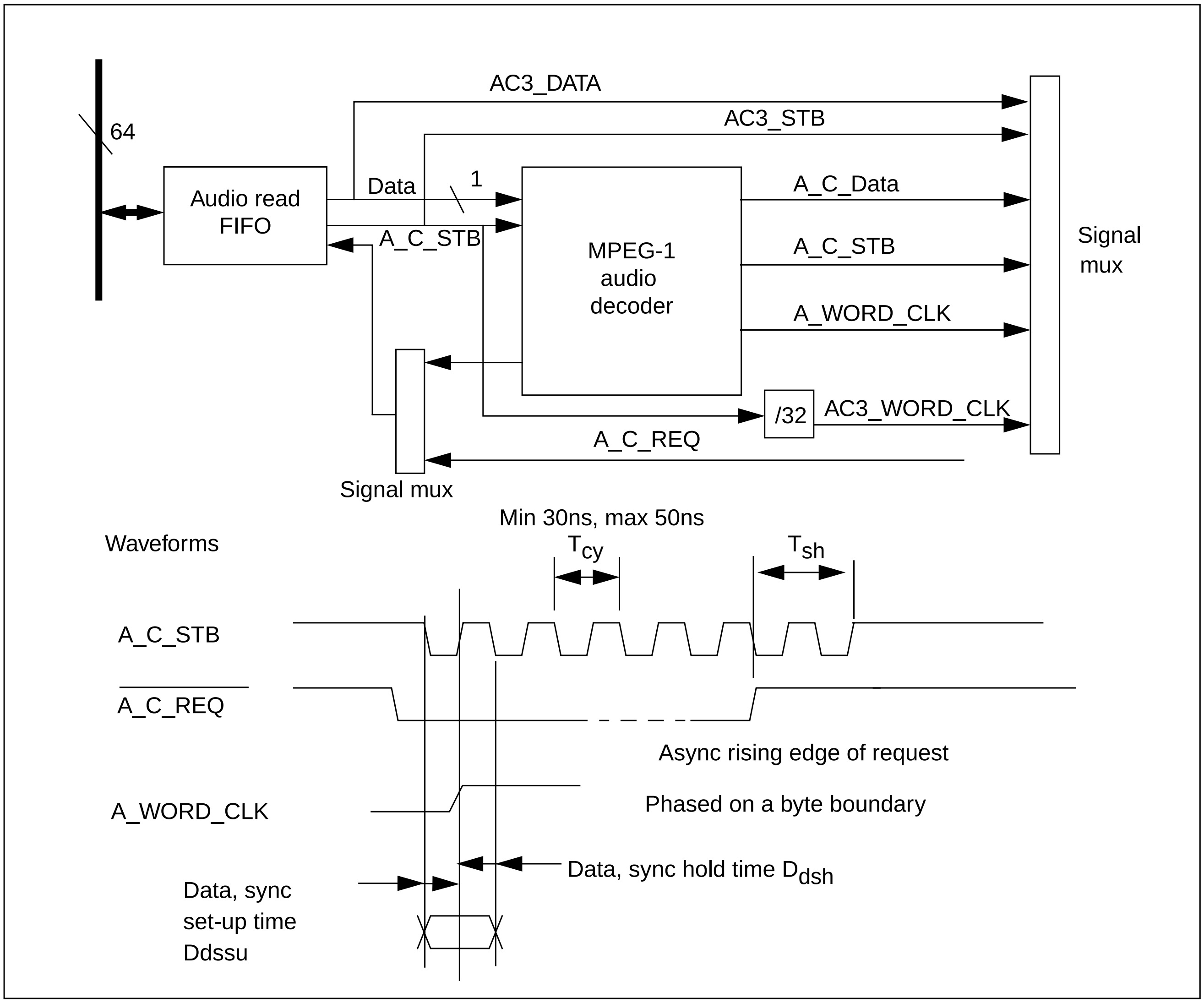


Figure 20.6 AC-3 interface detail

| Parameter | Minimum | Maximum | Units |
|---|---|---|---|
| Tcy | 30 | 50 | ns |
| Tcy | 66 | 100 | ns |
| Tsh | 0 | 2 | cycles |
| Ddsh | 15 | | ns |
| Ddssu | 15 | | ns |

Table 20.5  AC-3 interface timing

# Part D Display

# 21 Display functions

## 21.1 Overview

The graphics and display subsystem reads, processes, overlays and mixes pixel data stored in various buffers in SDRAM, and produces a combined image for display on a TV. The buffers are called the display planes.

The subsystem assumes three display planes as follows, overlaid in this order:

1. MPEG video plane (see section 21.2),
2. sub-picture plane (see section 21.3),
3. on-screen display plane (OSD) (see section 21.4).

The display planes are shown in Figure 21.1.

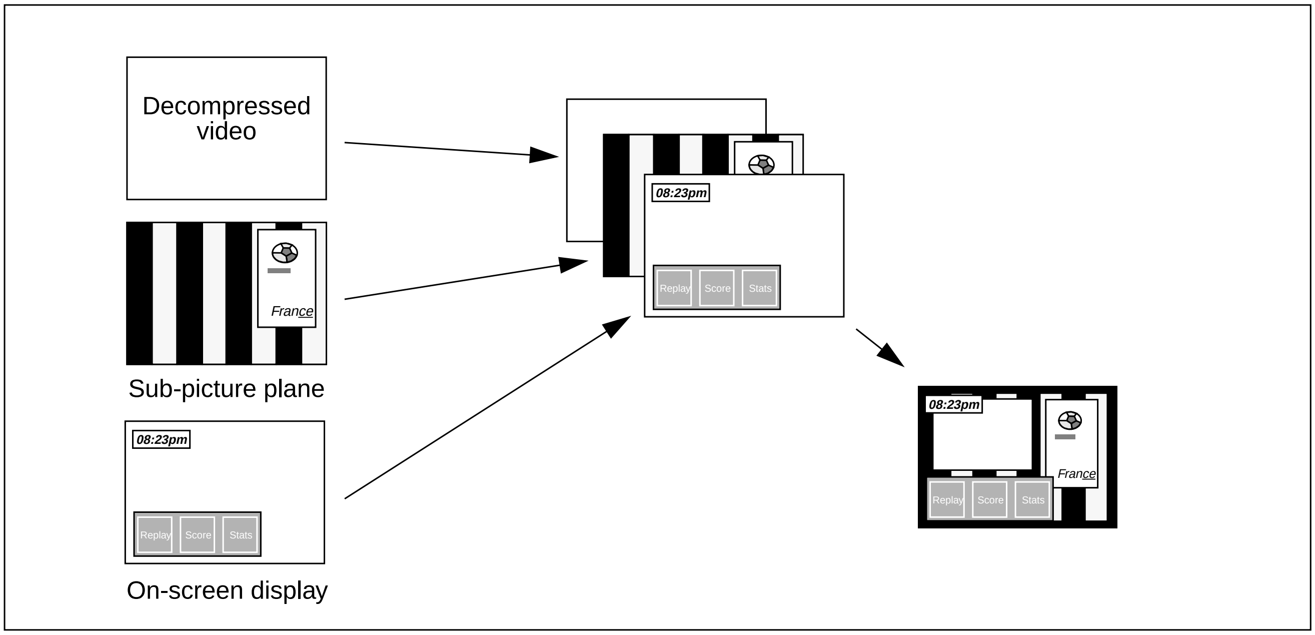


Figure 21.1 Display planes

Figure 21.2 is a simplified block diagram of the display unit. The mixing of the elements of the final picture is described in section 21.5.

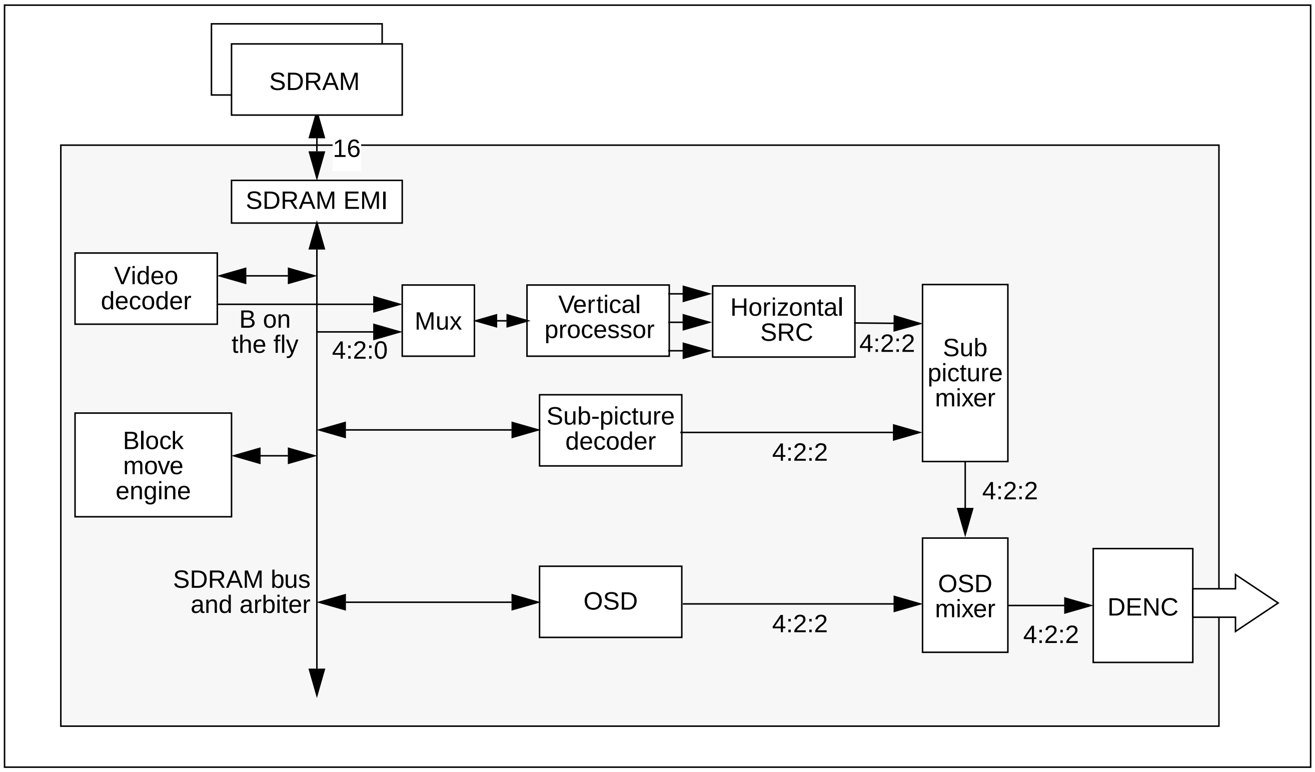


Figure 21.2 MPEG display architecture

## 21.2 MPEG video plane

The picture data is received either:

- from the display frame buffer area of the external memory, or
- directly from the MPEG video decoder in the case of B frames in memory reduction mode.

The data is passed through three FIFOs (one for luminance and two for chrominance) into the video post-processor. The video post-processor generates a line-based raster from the frame store, which is organized as MPEG macroblocks. It also performs the pan/scan operation and vertical filtering of the decoded video. The pan/scan operation is described in section 21.2.3 and the vertical filter is described in section 21.2.4.

The output of the video post-processor is fed to the sample rate converter (SRC). The SRC is an 8-tap filter, which has two functions:

- up and down scaling of pel data when the displayed line length is greater or smaller than the decoded picture width, and
- implementation of the fractional part of the pan-scan horizontal offset.

The outputs from the SRC are upsampled lines each having equal numbers of luminance and chrominance samples. The SRC can be bypassed if desired.

The sample rate converter is described in section 21.2.2.

### 21.2.1 Setting up the Display

The **VID_DFP** and **VID_DFC** registers must be set up with the base address of the buffer containing the picture to be displayed. This register is double-buffered; when a new value is written it is taken into account on the occurrence of a VSYNC. Thus it is possible to write a new value for this pointer every field, although it would normally be updated only once per frame.

The picture stored in the buffer is always treated as a frame by the STi5500. If at any time no display is required, bit **VID_DCF.EVD** may be reset, in which case a constant black value is output.

The size and location of the display window is defined by the registers **VID_XDO**, **VID_XDS**, **VID_YDO** and **VID_YDS**. The values loaded into these registers define the horizontal and vertical boundaries of the displayed picture, as shown in Figure 21.3.

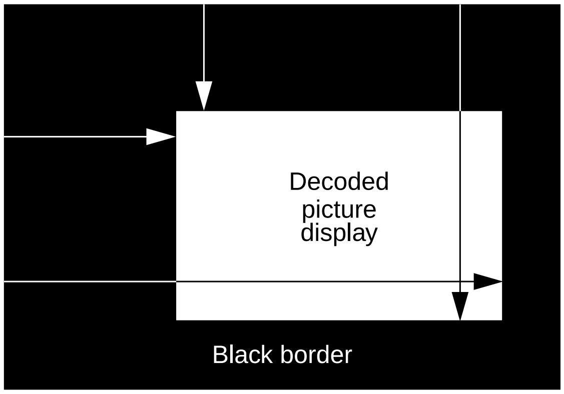


Figure 21.3 Display window positioning

The registers **VID_MCH** and **VID_MLU** must also be set up properly for display. The programming details can be found in the register manual.

Register **VID_YDO** is loaded with the number of the last line of the upper border, where lines are numbered in fields as shown in Figure 21.3. The first active line is therefore defined by:

First active line = **VID_YDO** + 1

The same **VID_YDO** value serves for both fields; the uppermost line of the picture display will be in the top field. Register **VID_YDS** is loaded with a number defining the last line of the picture display in a field, according to the relation:

Last active line = **VID_YDO** + (*vertical_size* / 2) = **VID_YDS** + 129

For example, with a 525/60 display, in which the vertical size of the decoded picture is 480 lines, typical values of **VID_YDO** and **VID_YDS** could be:

**VID_YDO** = 21, **VID_YDS** = **VID_YDO** + 240 - 129 = 132,

and with a 625/50 display, in which the vertical size of the decoded picture is 576 lines, typical values of **VID_YDO** and **VID_YDS** could be:

**VID_YDO** = 22, **VID_YDS** = **VID_YDO** + 288 - 129 = 181

Register **VID_XDO** defines the number of PIXCLK cycles between the falling edge of the signal HSYNC and the beginning of the picture display, according to the relation:

Cycles from HSYNC to start of picture = (2 x **VID_XDO**) + 40

The ITU-R 601 standard defines this number to be 264 27MHz clock cycles for a 625/50 display, and 244 for a 525/60 display. The respective values of **VID_XDO** are thus 112 and 102.

The first picture data in a line is always a CB component. It is output in the 2**VID_XDO** + 41st PIXCLK cycle after the falling edge of HSYNC.

Since the external video generation circuitry will usually relate its Y/C phasing to the horizontal synchronization signal, and has no knowledge of the value of **VID_XDO**, not all values of horizontal offset will be usable; some will cause incorrect interpretation of the color difference components. In any given system, **VID_XDO** values will have to be either always odd or always even.

**VID_XDS** is loaded with a number defining the last active sample in each line, counted in units of PIXCLK cycles from the falling edge of the signal HSYNC, according to the relation:

Last sample of active video = (2 x **VID_XDS**) + 28

Thus, if L is the number of pels per line of the displayed picture, then:

2**VID_XDO** + 40 + 2L = 2**VID_XDS** + 28, and thus

**VID_XDS** = **VID_XDO** + L + 6.

If L = 720, then **VID_XDS** = **VID_XDO** + 726.

The resolution to which the horizontal offset and end values can be defined is equal to two cycles of PIXCLK. The **VID_DCF.PXD** bit is used to position the display window horizontally to a finer precision. When this bit is set, the active video is delayed by one PIXCLK cycle. Since the first active video sample is $C_B$, the Y/C phasing with respect to the horizontal synchronization signal will change.

### 21.2.2 Sample rate converter

The purpose of the sample rate converter (SRC) is to allow up or down sampling of picture data in order to increase or decrease the number of horizontal samples in a line. Upsampling is necessary if the horizontal size of the display is greater than the decoded picture width. For example if it is required to display a 720-pel wide 16:9 source image on a 4:3 display also of 720-pel width, then 540 pels selected from each source line must be upsampled to 720. Downsampling is required when the resolution of the display is less than that of the decoded image. For example when square pixels are required for an NTSC image the 720 pixel wide image decoded must be downsampled to 640 pixels.

To enable the SRC, bit **VID_DCF.DSR** must be reset. If this bit is set, the SRC is bypassed and the horizontal resolution of the decoded picture is not changed. The sample rate converter can change the sampling rate by a programmable factor. The upsampling ratio is limited to 8 and the downsampling to less than or equal to a factor of 2. The same filter is used both for upsampling and downsampling. As either of these limits is approached artifacts may appear in the displayed image. The SRC operates by directly interpolating samples required for the new sampling rate by using those of the decoded picture data read from the display buffer. This is performed by an 8-tap interpolation filter with the structure shown in Figure 21.4.

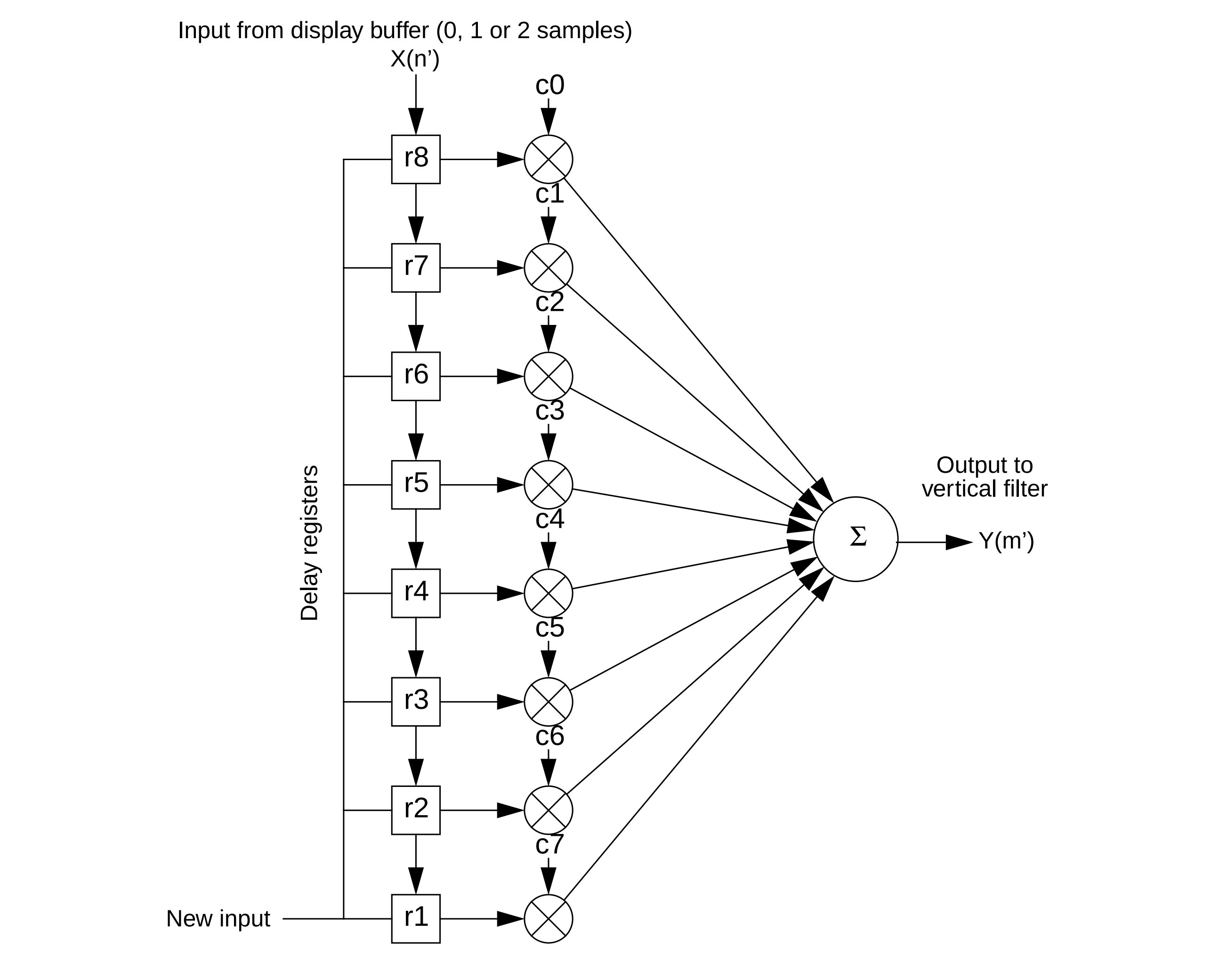

Input from display buffer (0, 1 or 2 samples)
X(n')
c0
c1
c2
c3
c4
c5
c6
c7
r8
r7
r6
r5
r4
r3
r2
r1
Delay registers
New input
Σ
Output to vertical filter
Y(m')

Figure 21.4 8-tap interpolation filter

The filter has three sets of delay registers multiplexed between the Y, $C_B$ and $C_R$ samples. It has 8 sets of coefficients, each set defining one of 8 sub-pel interpolation positions. Consider an upsampling example, for sub-pel position 0, the output is aligned with stored sample “r4”, for sub-pel position 1, the output corresponds to an interpolated pel position one eighth of the distance from sample “r4” to sample “r5”, and so on. The number of inputs clocked into the SRC is equal to the number of samples used in each line of the source image, and the number of outputs generated is equal to the number of samples displayed. Thus the rate of generation of outputs will be greater than the input data rate in the case of upsampling and less in the case of downsampling.

### Operation of the SRC

The sample rate converter works in the following manner: The SRC takes block of M samples of the input signal denoted as x(n'), n' = 0,1,2,3, . . . . M-1. and computes a block of L output samples y(m'), m' = 0,1,2, .. L-1.

For each output sample time m', m' = 0,1,2 . ., L-1 the 8 samples in the filter are multiplied with one of the 8 sets of filter coefficients the products are accumulated to give the output y(m'). Each time the quantity m'M/L increases by one, one sample from the input buffer is shifted into the filter.

The coefficient set used will depend on the position of the sample being generated relative to the original samples of the source image. Thus after L output values are computed M input samples have been shifted into the filter delay registers.

The SRC up/down sampling factor is set up in the **VID_LSR** register. The re-sampling factors for the luminance and chrominance components are exactly the same. The resampling factor is equal to L/M. The value programmed into **VID_LSR** is 256 × M/L. This value is used to determine both the rate of input of data into the filters and the sequence of sub-pel interpolation positions. The mechanism by which this is achieved is shown in Figure 21.5.

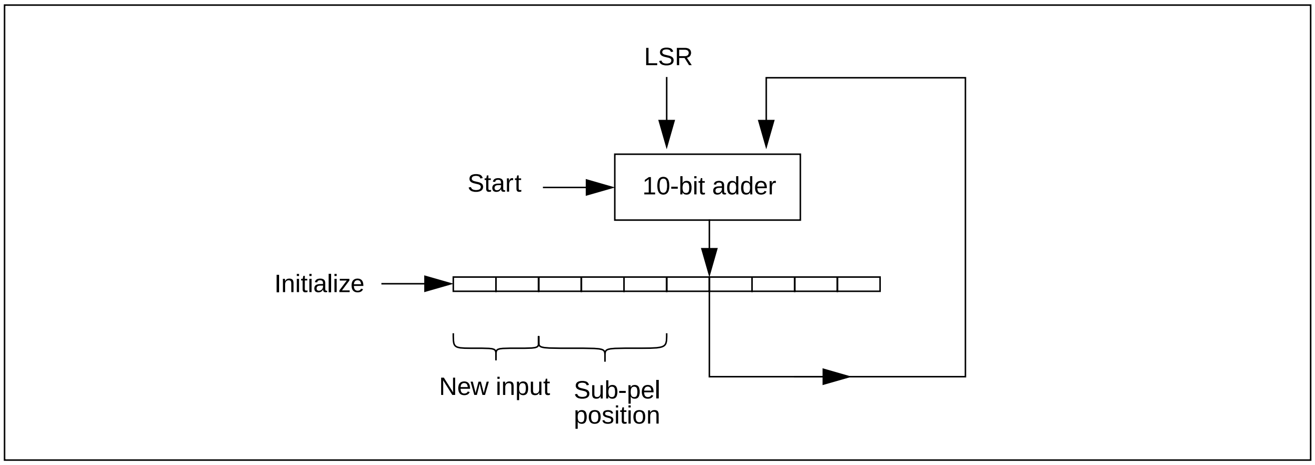
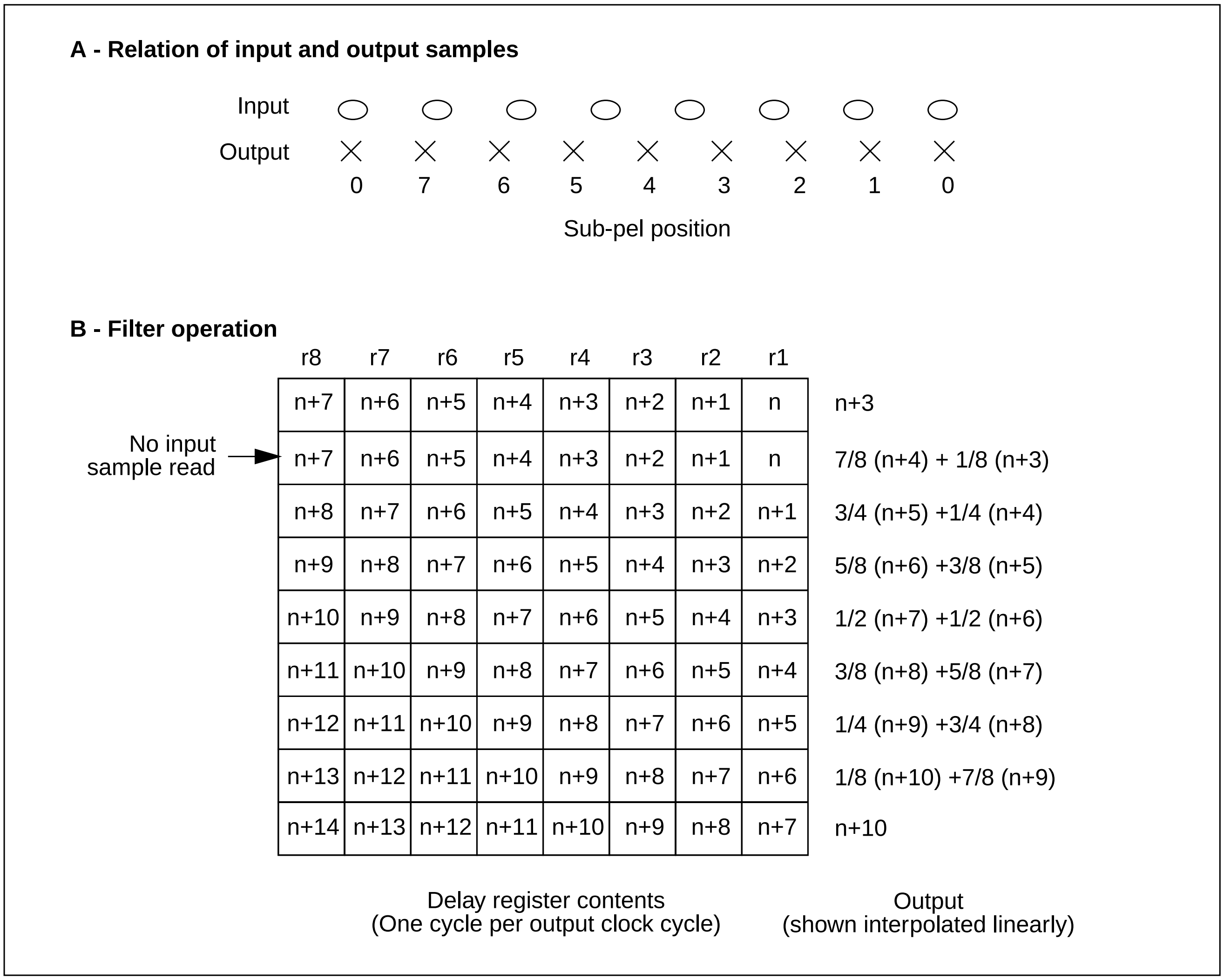


Figure 21.5 Up/down sampling filter control

### Upsampling example

The example in Figure 21.6 illustrates the operation of the sample rate converter when the upsampling ratio is 8:7. For every 8 samples clocked out of the filters, 7 samples are clocked in.

To illustrate the interpolation positions, at the right of Figure 21.6 are shown the outputs which would occur with a simple linear interpolation (i.e.a 2-tap filter). The actual SRC output values are the 8-tap filter outputs with coefficients appropriate to sub-pel positions 0, 7, 6, 5, 4, 3, 2, 1, 0 etc. The SRC output is limited to lie within the range [1,254], so the codes 0x00 and 0xFF are never output, giving compatibility with ITU-R 656.



| r8 | r7 | r6 | r5 | r4 | r3 | r2 | r1 | |
|---|---|---|---|---|---|---|---|---|
| n+7 | n+6 | n+5 | n+4 | n+3 | n+2 | n+1 | n | n+3 |
| n+7 | n+6 | n+5 | n+4 | n+3 | n+2 | n+1 | n | 7/8 (n+4) + 1/8 (n+3) |
| n+8 | n+7 | n+6 | n+5 | n+4 | n+3 | n+2 | n+1 | 3/4 (n+5) +1/4 (n+4) |
| n+9 | n+8 | n+7 | n+6 | n+5 | n+4 | n+3 | n+2 | 5/8 (n+6) +3/8 (n+5) |
| n+10 | n+9 | n+8 | n+7 | n+6 | n+5 | n+4 | n+3 | 1/2 (n+7) +1/2 (n+6) |
| n+11 | n+10 | n+9 | n+8 | n+7 | n+6 | n+5 | n+4 | 3/8 (n+8) +5/8 (n+7) |
| n+12 | n+11 | n+10 | n+9 | n+8 | n+7 | n+6 | n+5 | 1/4 (n+9) +3/4 (n+8) |
| n+13 | n+12 | n+11 | n+10 | n+9 | n+8 | n+7 | n+6 | 1/8 (n+10) +7/8 (n+9) |
| n+14 | n+13 | n+12 | n+11 | n+10 | n+9 | n+8 | n+7 | n+10 |

Figure 21.6 SRC example for 8:7 upsampling

The **VID_LSR** value is added into an accumulator register at a rate equal to the filter output rate. The top two bits indicate how many new inputs are to be loaded into the filter (0,1 or 2). The next three bits of the accumulator register are used to select the sub-pel position. For example, with an upsampling factor of 8:7, the **VID_LSR** value is (256/8) × 7 = 224. The sequence of values in the accumulator register will be as shown in Table 21.1, assuming that it is initialized to zero.

| Accumulator register | New input | Sub-pel position |
|---|---|---|
| 0 | yes | 0 |
| 224 | no | 7 |
| 192 | yes | 6 |
| 160 | yes | 5 |

Table 21.1 Accumulator register sequence for upsampling example

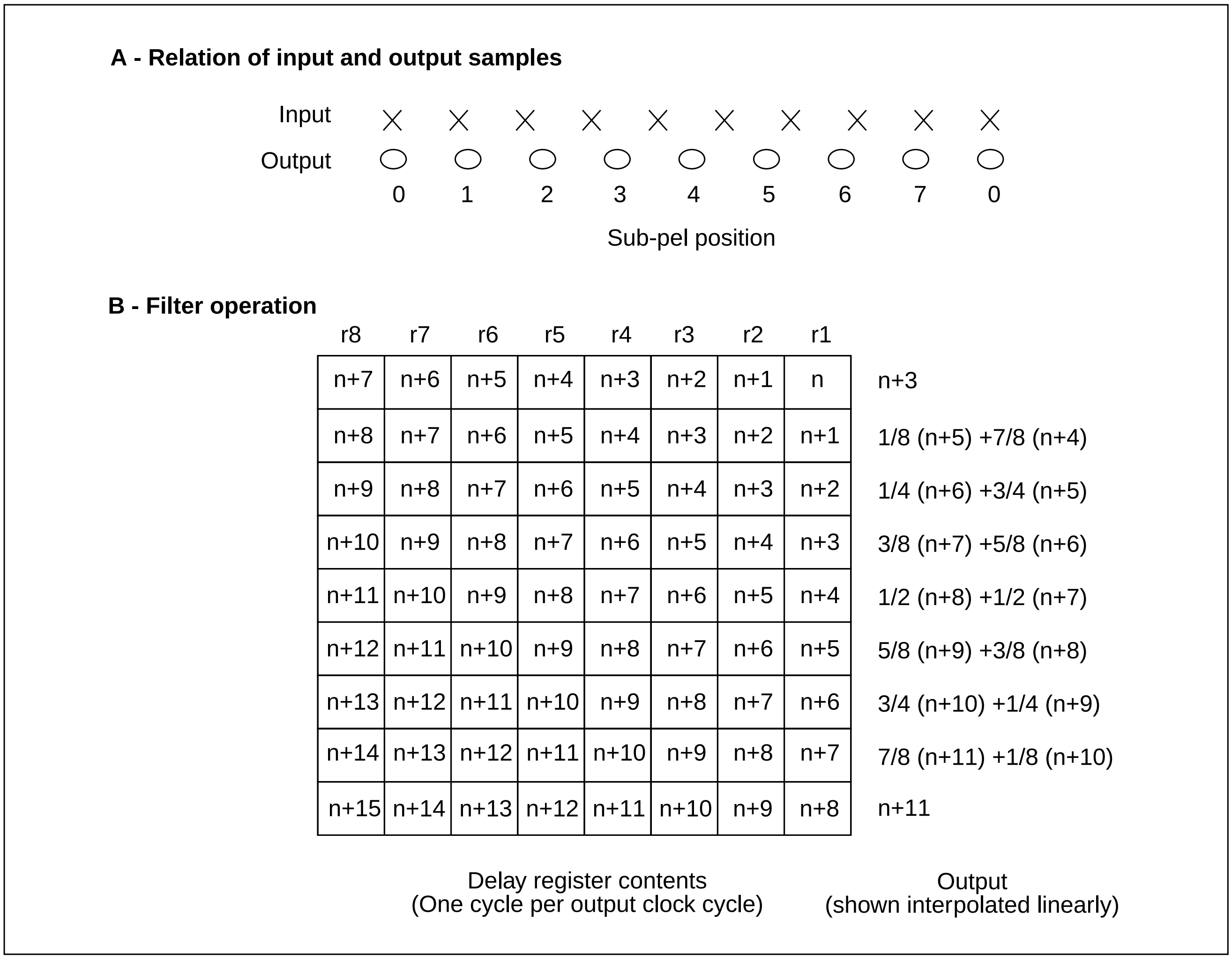
| Accumulator register | New input | Sub-pel position |
|---|---|---|
| 128 | yes | 4 |
| 96 | yes | 3 |
| 64 | yes | 2 |
| 32 | yes | 1 |
| 0 | yes | 0 |

Table 21.1 Accumulator register sequence for upsampling example

The **VID_LSR** value thus defines a cycle of sub-pel positions as well as the rate of data input. If a value of less than 32 is loaded into **VID_LSR**, i.e. an upsampling ratio of greater than 8 is defined, there could be repeated values in the filter output. This may cause unacceptable display artifacts.

### Downsampling example

The example shown in Figure 21.7 illustrates the operation of the sample rate converter when the downsampling ratio is 9:8 (720:640).



| r8 | r7 | r6 | r5 | r4 | r3 | r2 | r1 | |
|---|---|---|---|---|---|---|---|---|
| n+7 | n+6 | n+5 | n+4 | n+3 | n+2 | n+1 | n | n+3 |
| n+8 | n+7 | n+6 | n+5 | n+4 | n+3 | n+2 | n+1 | 1/8 (n+5) +7/8 (n+4) |
| n+9 | n+8 | n+7 | n+6 | n+5 | n+4 | n+3 | n+2 | 1/4 (n+6) +3/4 (n+5) |
| n+10 | n+9 | n+8 | n+7 | n+6 | n+5 | n+4 | n+3 | 3/8 (n+7) +5/8 (n+6) |
| n+11 | n+10 | n+9 | n+8 | n+7 | n+6 | n+5 | n+4 | 1/2 (n+8) +1/2 (n+7) |
| n+12 | n+11 | n+10 | n+9 | n+8 | n+7 | n+6 | n+5 | 5/8 (n+9) +3/8 (n+8) |
| n+13 | n+12 | n+11 | n+10 | n+9 | n+8 | n+7 | n+6 | 3/4 (n+10) +1/4 (n+9) |
| n+14 | n+13 | n+12 | n+11 | n+10 | n+9 | n+8 | n+7 | 7/8 (n+11) +1/8 (n+10) |
| n+15 | n+14 | n+13 | n+12 | n+11 | n+10 | n+9 | n+8 | n+11 |

Delay register contents
(One cycle per output clock cycle)

Output
(shown interpolated linearly)

Figure 21.7 SRC example for 9:8 downsampling

The **VID_LSR** value required for a downsampling ratio of 9:8 is 256 x 9 /8 = 288.

| Accumulator register | Number of inputs | Sub-pel position |
|---|---|---|
| 0 | 1 | 0 |
| 288 | 1 | 1 |
| 576 | 1 | 2 |
| 864 | 1 | 3 |
| 128 | 1 | 4 |
| 416 | 1 | 5 |
| 704 | 1 | 6 |
| 992 | 1 | 7 |
| 0 | 2 | 0 |

Table 21.2 Accumulator register sequence for downsampling example

At the start of a line, the 3 sets of delay registers r1, r2 and r3 are loaded with the black value (Y=16, $C_B$=$C_R$=128).

The first output is thus derived from the inputs stored in registers r4 to r8. At the end of a line, the last eight input samples are stored in registers r1 to r8.

The last valid interpolation is between the samples stored in r4 and r5. Correct Interpolation is not possible beyond this except in the case where the next output is in sub-pel position 0. This output is valid since coefficient C0 is zero for this position and the invalid sample beyond the end of the line is ignored.

There is thus no valid interpolation possible between the last four input samples. This is illustrated in Figure 21.8 in which 544 pels are upsampled to 721, in which the upsampling ratio is 4:3. The **VID_LSR** register would be loaded with the value 192.

The number of valid outputs generated can be calculated as follows:

The ratio between the number of input and output samples is 256:**VID_LSR**. Given that the last output sample cannot occupy a position beyond the fourth-last input sample, the following inequality is always true:

**VID_LSR** (N-1) ≤ 256 (M-4)

where N is the number of output samples and M is the number of input samples. The value of N is thus given by:

N = ⌊256 (M-4) / **VID_LSR** + 1⌋

where ⌊x⌋ indicates the integer part of x.

The value programmed into the **VID_XDS** register must be such that all samples beyond the last valid one are masked.

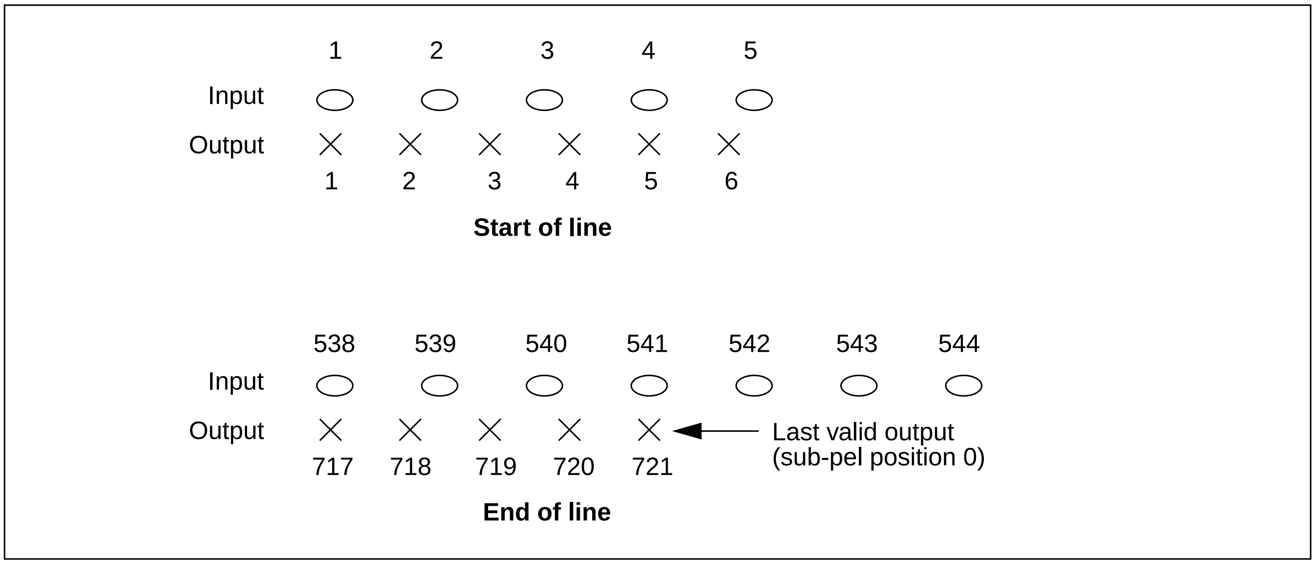


Figure 21.8 Upsampling from 544 to 720

### 21.2.3 Pan/scan vectors

When the display window has a smaller horizontal dimension than the decoded picture, a vector can be programmed in order to define the starting point of the displayed picture, as shown in Figure 21.9. The vertical component must be macroblock aligned, so the line number must be a multiple of 16.

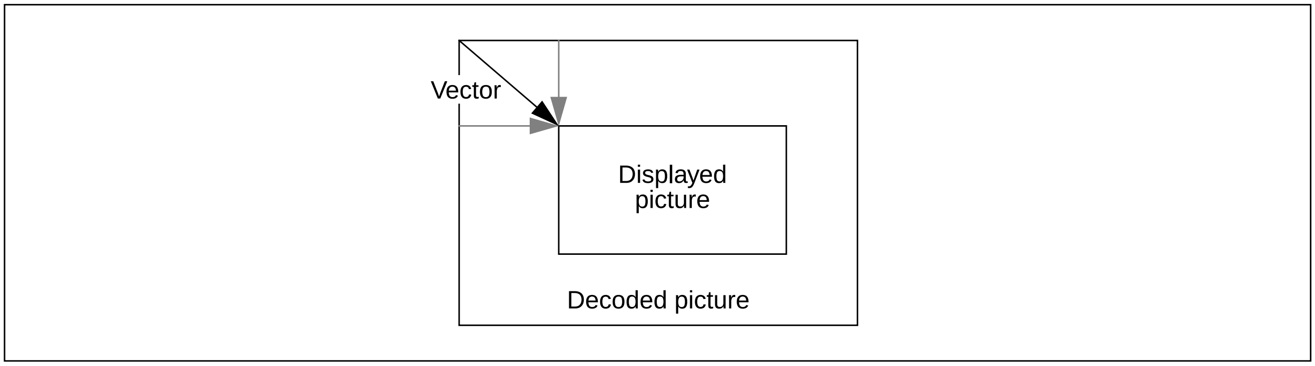


Figure 21.9 Pan/scan vector

This vector defines the point in the decoded picture which corresponds to the top-left-hand corner of the displayed picture. The displayed picture size and location is defined by the numbers programmed in registers **VID_XDO**, **VID_XDS**, **VID_YDO** and **VID_YDS**.

The pan/scan vector components are programmed into the registers **VID_PAN**, **VID_LSO** and **VID_CSO**. These registers are double-buffered; when a new value is written it is taken into account on the occurrence of a VSYNC. Thus it is possible to write a new value of the pan/scan vector for every field.

The integer part of the horizontal component of the pan/scan vector is loaded into the **VID_PAN** register, and the fractional part defines the contents of the **VID_LSO** and **VID_CSO** registers. The relationship between these quantities is illustrated in Figure 21.10.

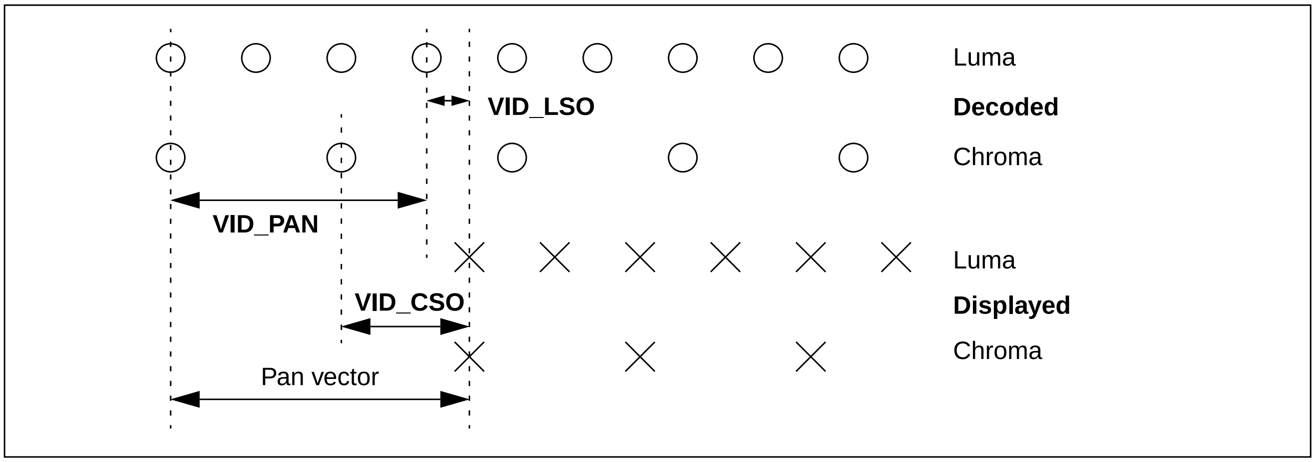


Figure 21.10 Components of the pan/scan vector

The numbers loaded into the **VID_LSO** and **VID_CSO** registers are used to initialize the luminance and chrominance upsampling control registers at the start of every line. **VID_LSO** is set up directly with the value of the fractional part of the pan/scan vector horizontal component. **VID_CSO** is set up with half of this number, plus 128 if the integer part is an odd number. The resolution to which the horizontal component can be defined is 1/8 pel.

The STi5500 does not support vertical pan/scan components.

### 21.2.4 Vertical filter

Vertical processing is used for the following purposes:

- to double the number of chroma lines (4:2:0 to 4:2:2);
- for letterbox conversion (4 line to 3 conversion);
- to display half-resolution images on a full resolution display.

The STi5500 supports only intra-field vertical processing.

Table 21.3 lists the filter modes. The modes are shown in more detail in the figures which follow.The different filter modes are programmed by writing the mode number in the bit field **VFC[2:0]** in the **VID_DCF** register.

| Mode | Decode | Display |
|---|---|---|
| 0 | Full resolution | Full resolution display with chrominance interpolation |
| 1 | Full resolution | Full resolution display with chrominance line repeat |
| 2 | Half resolution | Full resolution display with luma interpolation and chroma line repeat |
| 3 | Full resolution | Letterbox filtering |
| 4 | Half resolution | Letterbox filtering |

Table 21.3  Vertical filtering modes

## Mode 0

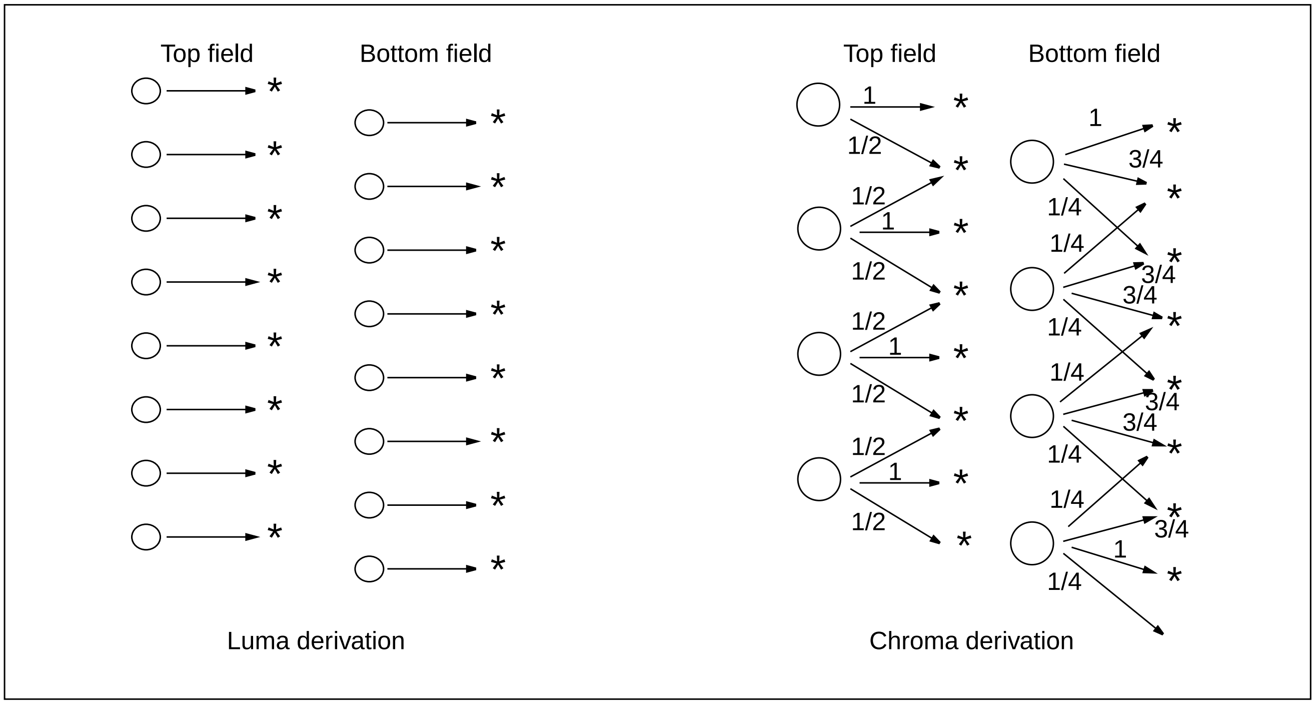


Figure 21.11 Mode 0 filtering

## Mode 1

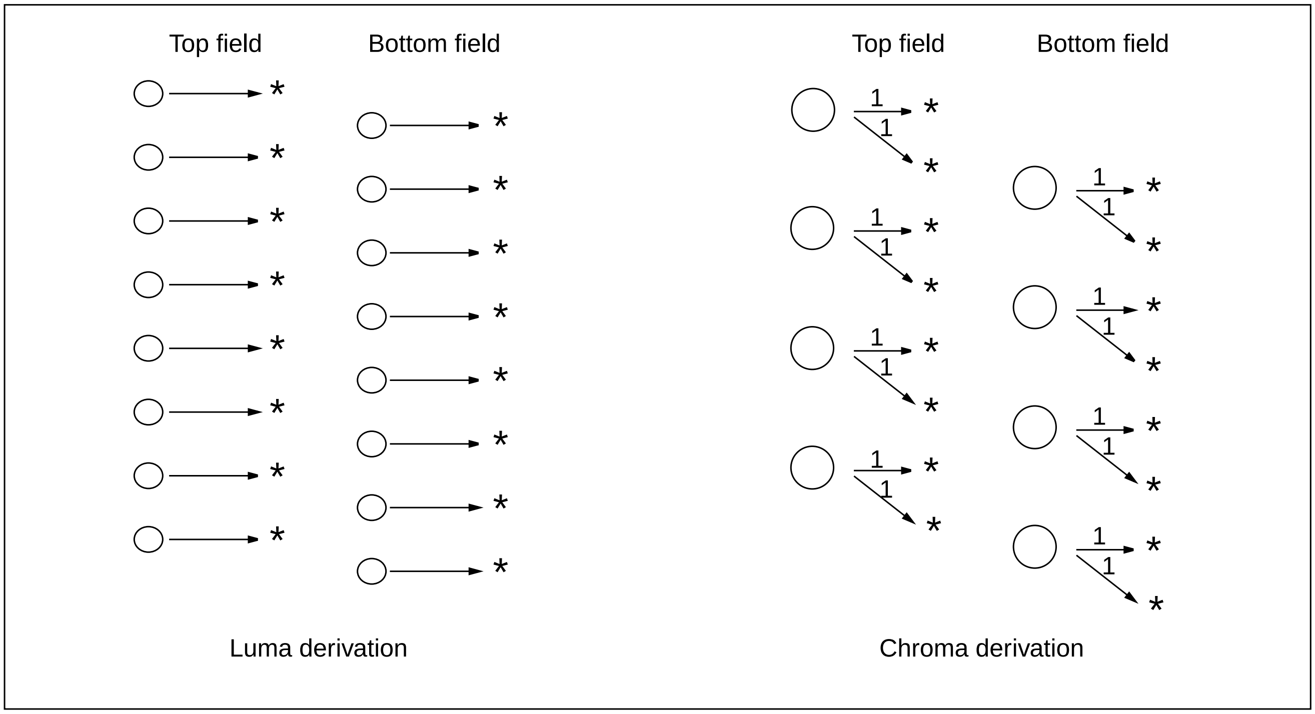


Figure 21.12 Mode 1 filtering

## Mode 2

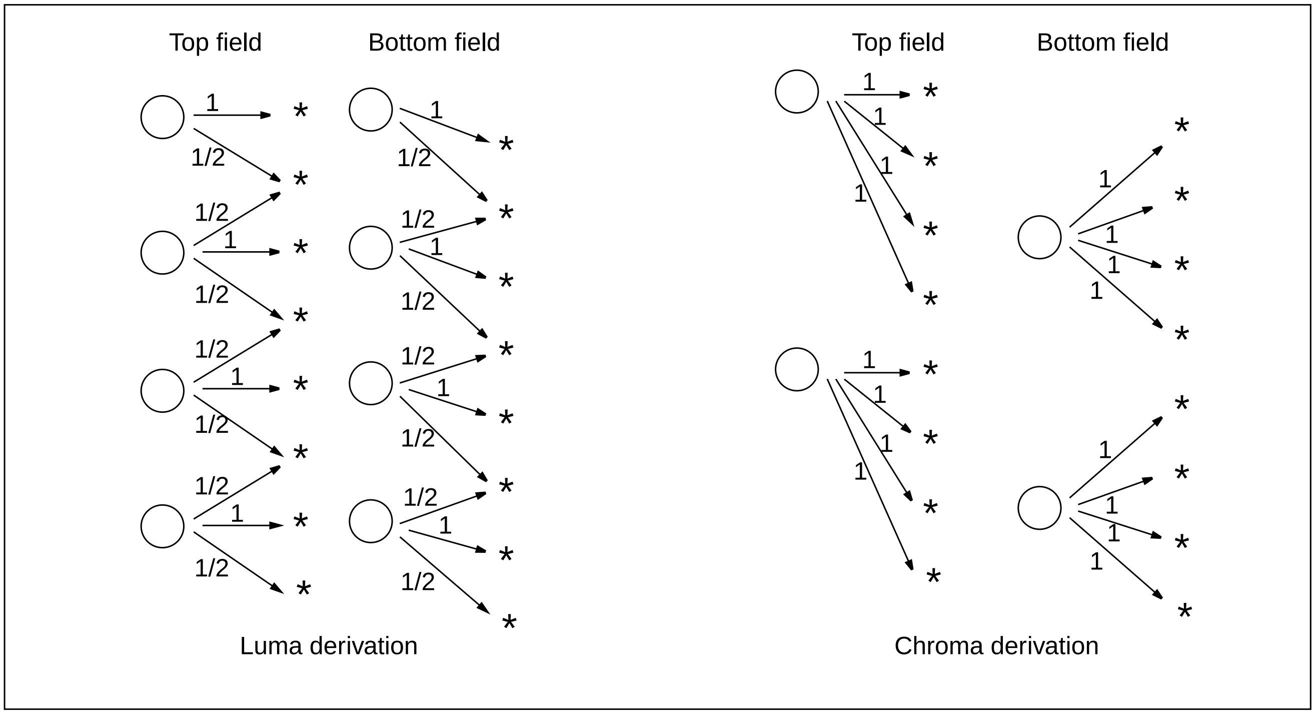


Figure 21.13 Mode 2 filtering

## Mode 3

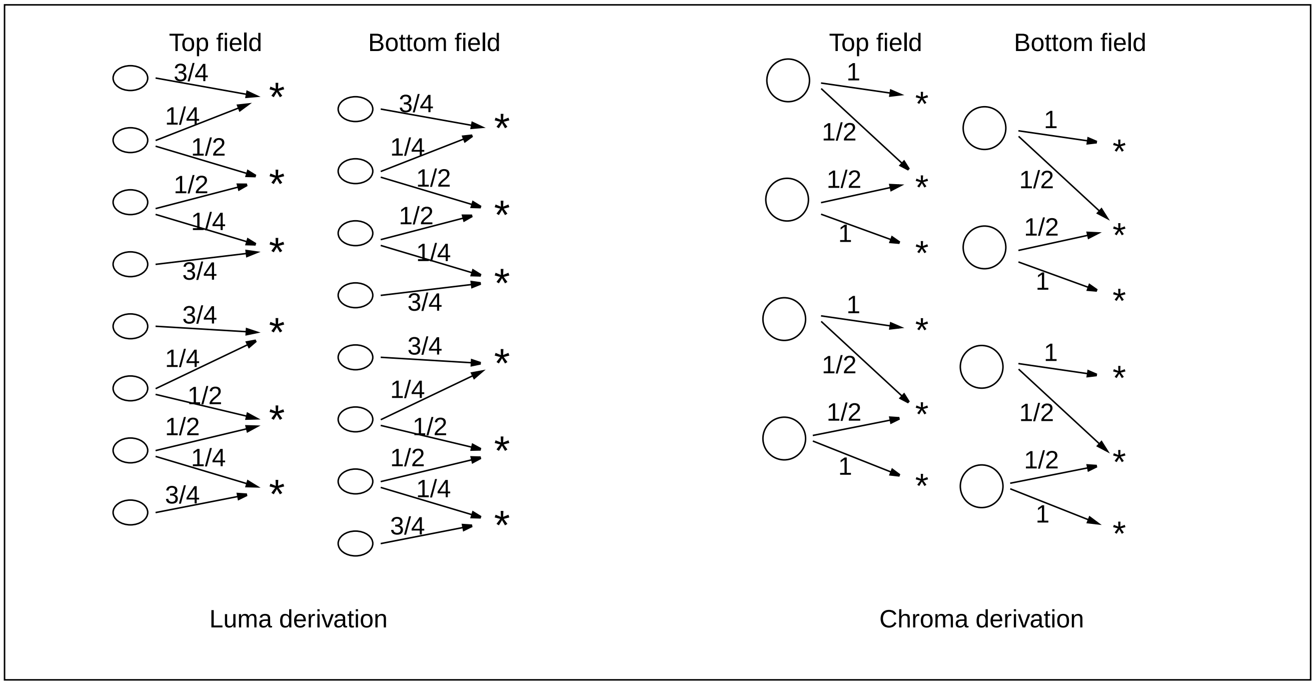


Figure 21.14 Mode 3 filtering

### Mode 4

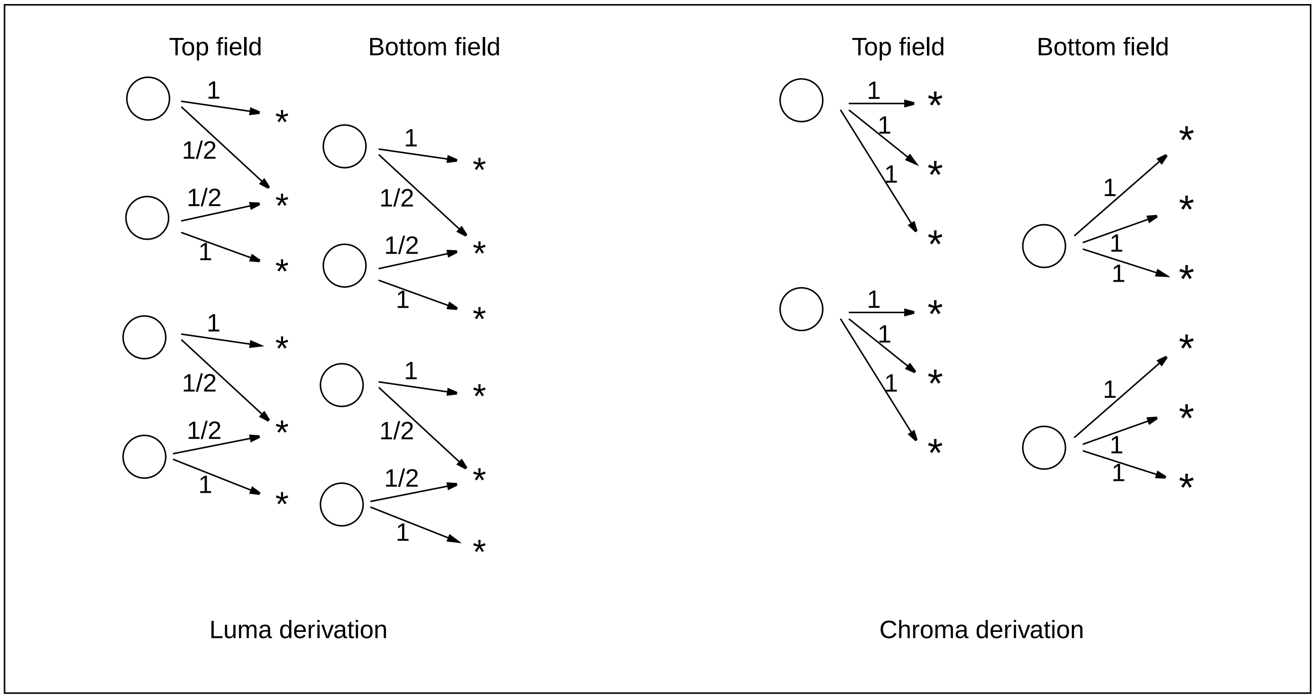


Figure 21.15 Mode 4 filtering

### Vertical Filter Precision

The vertical filter calculation is performed in unsigned arithmetic with full precision and then the 10-bit results are rounded to 8 bits for output. Different rounding rules are used for luminance and chrominance. The rules are as follows:

- For luminance, the results are rounded towards zero, i.e. if the bottom two bits are 00, 01 or 10, the 10-bit number is truncated to 8, while if the bottom two bits are 11, one is added to the truncated 8-bit number.
- For chrominance, the results are rounded towards 128, i.e. if the 10-bit number is larger than $1000000000_2$, the rule above is applied, while if the number is less than this, $10_2$ is added to the 10-bit number before truncation.

### 21.2.5 Degradation mode

In certain situations the system constraints may justify use of the STi5500 in a configuration where the available bandwidth on the SDRAM interface is limited. There could be many reasons for these constraints, such as a low clock frequency to use cheaper SDRAMS, or the processor making heavy use of the SDRAM. MPEG decode and display, being a real-time process and also a heavy user of SDRAM memory bandwidth will then require a graceful degradation mode.

A small piece of hardware is implemented in the decoder to measure the effective distance (in pixels) between the display process and the decode process. Under conditions of limited bandwidth the decoder will become late and therefore may get caught by the real-time limited display process.

Degradation mode can be enabled and disabled using the register **VID_PTH**. A threshold or minimum allowable distance between the decode and display processes can also be set. If this threshold is crossed the decoder will automatically insure that any bidirectionally predicted macroblock access will result in only a single prediction access to external memory thus reducing the bandwidth required by the decoder and allowing recovery.

## 21.3 Sub-picture plane

The sub-picture plane displays the output from the sub-picture decoder, described in Chapter 19. The sub-picture is between the video plane and the OSD.

## 21.4 On-screen display (OSD)

The STi5500 has an integrated On-Screen Display (OSD) unit. This can be used to overlay the video picture with graphics generated by software. The display priority puts the OSD in front of the MPEG video mixed with the sub-picture. The OSD can be enabled or disabled. The OSD bit-map is defined with respect to the display area and is independent of the decoded picture size and any pan/scan offset. The output from the OSD is in 4:2:2 format.

The OSD of the STi5500 has the following special features:

- Linked list memory management;
- Selectable 2, 4 or 8-bits per pixel palette modes giving 4, 16 or 256 palette colors;
- 6-bit luma resolution and 4-bit chroma resolution per component;
- Programmable 4-bit mixing factor for each OSD region to blend the video plane and OSD data;
- Half resolution mode.

These features are described in the following sections.

The OSD unit uses color *look-up-tables* (LUTs), also called *palettes*, with 2-bit, 4-bit or 8-bit input. The LUT means that memory is used efficiently when only a few colors are needed. A 2-bit LUT means that four colors can be used at once, and each pixel of the bit-map occupies only two bits of memory. A 4-bit LUT gives 16 colors and an 8-bit LUT gives 256 colors. The palette of 4, 16 or 256 predefined colors is loaded into the SDRAM by software using the shared memory interface. The palette modes are described in section 21.4.6.

The output from the LUT is in the form of 14-bit pixels (6-bit Y, 4-bit Cb, 4-bit Cr) plus one bit for transparency control. The color modes are described in section 21.4.6.

The OSD can consist of a number of display regions, each with its own palette and characteristics. The number of OSD regions resident in memory at any time is limited only by the amount of memory available. Each region has a specification, stored in memory, which contains a header, possibly including a palette, and a bit-map. The specifications for the regions are linked in a list structure. The bit-map data in each specification is contiguous with the palette information, as shown in Figure 21.20. The bit-map refers to the 2-, 4- or 8-bit color definitions in the palette to create the required picture.

During the display of an image a small state machine first picks up the palette from SDRAM and loads it into the LUT then the OSD region start and stop addresses are read. When the display reaches the OSD start position (defined in the bit-map) the bit-map is sent pixel by pixel to the LUT and the display switches from video to the output of the LUT or a mixture of both.

This process continues until the defined stop position. Thus, for the defined OSD region, the video display is overlaid by the colors which are defined by a combination of the LUT and the bit-map.

### 21.4.1 Using the OSD

The OSD is enabled if bit **VID_DCF.EOS** is set. The starting address in memory of the OSD specification for the top field is defined by register **VID_OTP**, and that for the bottom field is defined by register **VID_OBP**.

The line numbers used to define the top and bottom of an OSD region are the internal (field) line numbers defined in Figure 21.16. It is thus possible to share the same OSD specification for both fields of a frame. In this case the **VID_OTP** and **VID_OBP** registers would be loaded with the same address.

OSD specifications can be written into the SDRAM using the ST20 or the block move DMA. They can be rapidly moved within SDRAM using the SDRAM block move function.

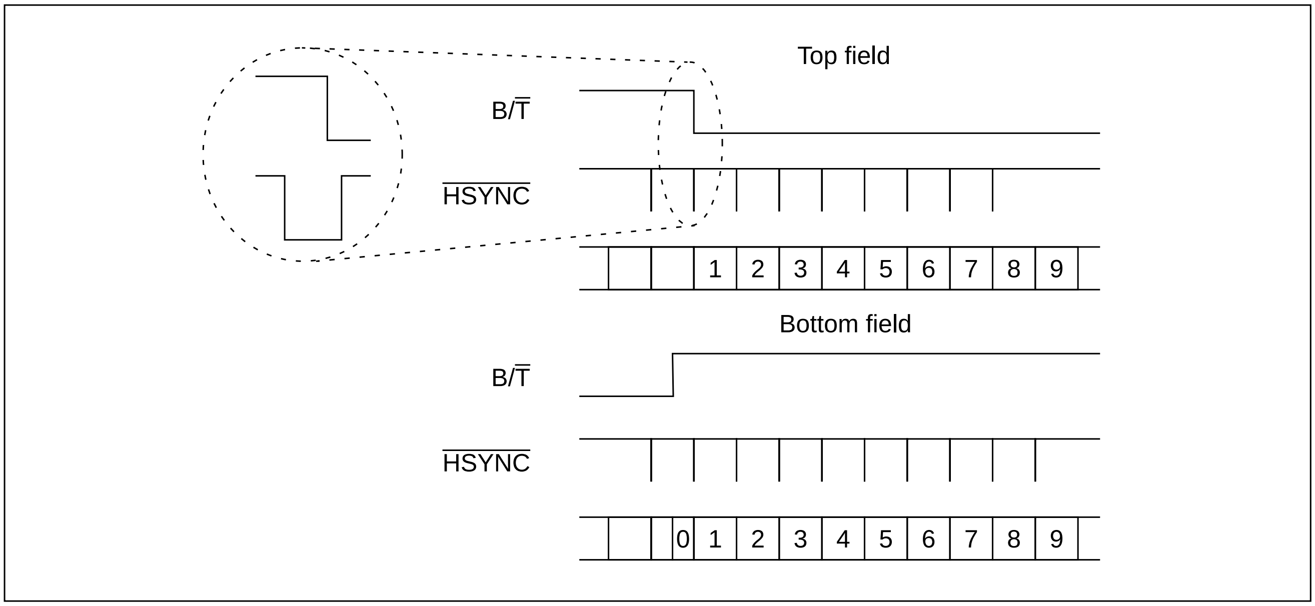


Figure 21.16 Internal line numbering

### 21.4.2 OSD regions

The OSD function can be used to display a user-defined bit-map over any part of the displayable (i.e. non-blanked) screen, independent of the size and location of the active video area (defined by **VID_XDO**, **VID_XDS**, **VID_YDO**, **VID_YDS**). This bit-map can be defined independently for each field.

The OSD consists of one or more regions in the display. Each region is a rectangle, and can have its own palette and other properties. Figure 21.17 shows examples of OSD regions. Region 3 shows that the OSD can be outside the active video area.

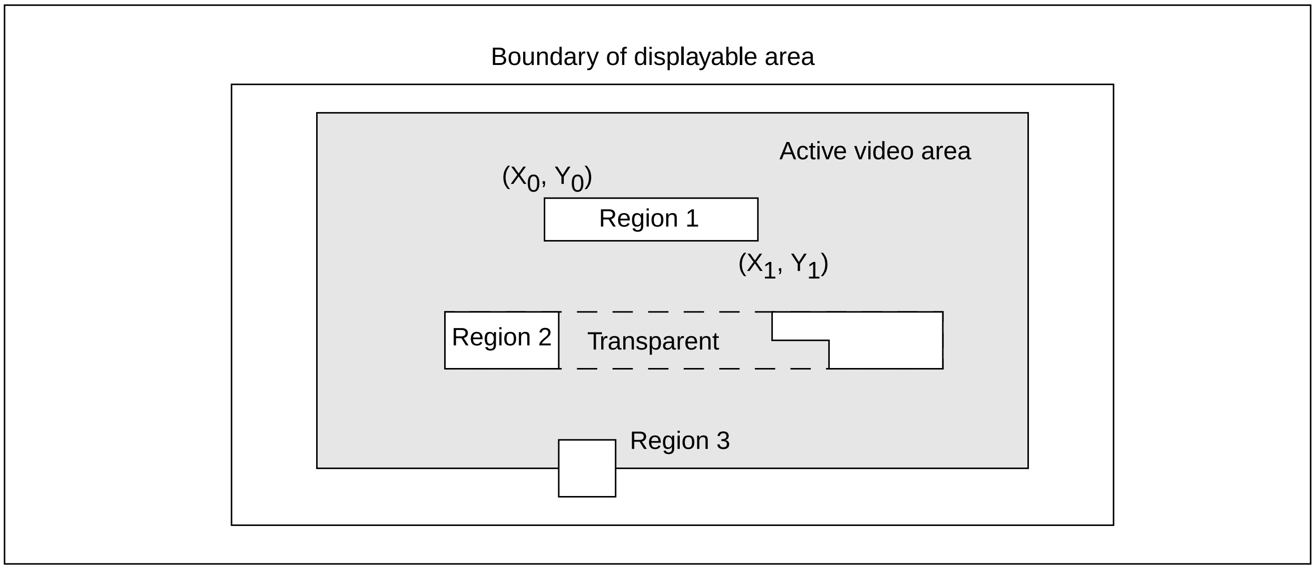


Figure 21.17 OSD regions

No display line can be included in more than one active OSD region, so only one OSD region can be active on a line. If two areas of OSD are required which include the same display line then one region must be defined which includes both areas. For example, Figure 21.18 shows two areas, marked A and B, with some display lines used in both areas. If these areas are to be active at the same time then one region, marked C, must be defined, which includes both areas. The area of C outside A and B can be defined as transparent.

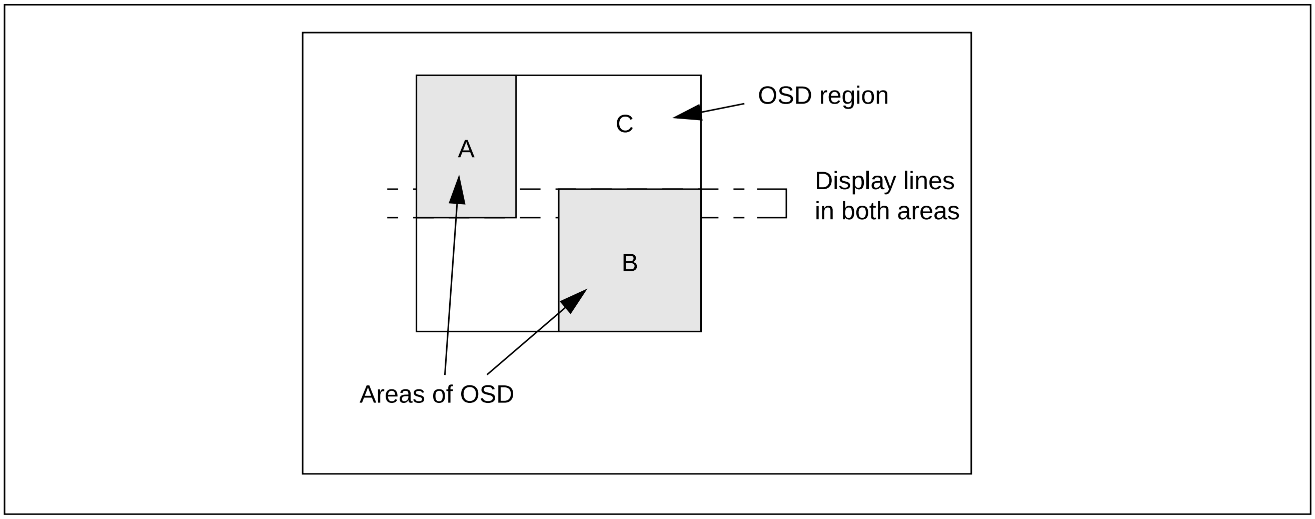


Figure 21.18 Two display areas using the same display lines

### 21.4.3 OSD specification

An OSD specification is two linked lists of blocks of 64-bit words, stored in SDRAM. One list is for the top field and one for the bottom, as shown in Figure 21.19. The order of the blocks in the list is the order of the regions from top to bottom of the display. The last block in an OSD specification must point to either:

- a null header line, which is a 64-bit line filled with zeros, or
- an invalid header line, which gives a starting line beyond the displayable area.

A null header line or an invalid header line will end the OSD.

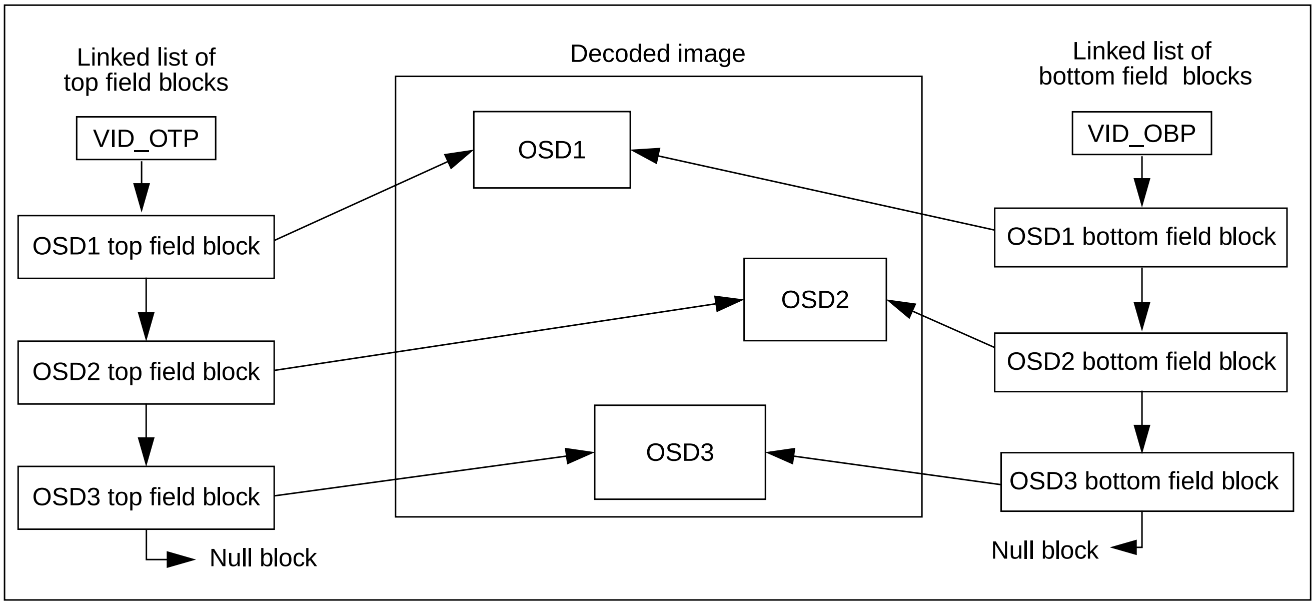


Figure 21.19 Linked list structure for OSD data

Each block defines one field of one region and includes a header, an optional palette and a bit-map. Each block must be aligned on a 32-bit boundary and the first block of each field must be aligned on a 128-bit boundary. Figure 21.20 shows a linked list of two OSD blocks.

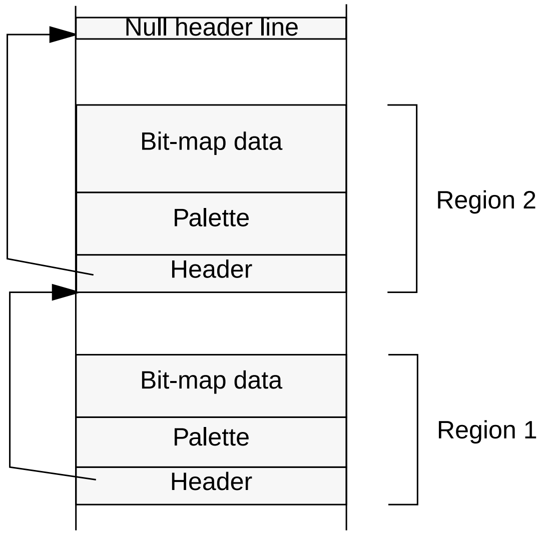


Figure 21.20 OSD specification

Each region has associated with it a palette defining 4, 16 or 256 colors, used by the bit-map. If required, one of these colors can be “transparent”, allowing the background to show through. Each region may have its own palette, or if a sequence of regions uses the same palette then the palette need only be defined in the first region of the sequence.

The header of each block contains a definition of the boundaries of the region, a pointer to the next region and other control information. The format of the palette depends on the palette mode, as described in section 21.4.6. The formats are given in section 21.4.8.

### 21.4.4 OSD region position

The position of each region of the OSD is defined in the header of the specification block. The positions of the left and right edge samples of an OSD region are defined as follows, in units of PIXCLK cycles from the falling edge of $\overline{\text{HSYNC}}$:

*left edge position* = (2 × *X_left*) + 9

*right edge position* = (2 × *X_right*) + 10

where *X_left* and *X_right* are the values defined in the header of the OSD region specification. This is illustrated in Figure 21.21.

*X_left* must always have the same parity as the offset loaded into the **VID_XDO** register (i.e. both must be even or both must be odd). This constraint ensures that the OSD region data samples are always correctly phased with respect to the active video. The first sample output in an OSD region is always a $C_B$ value.

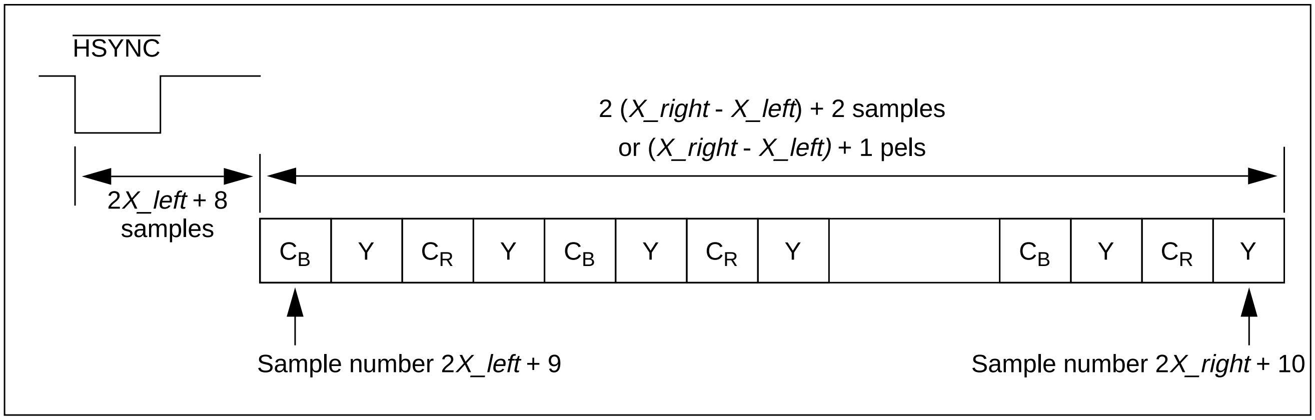

HSYNC
2 (X_right - X_left) + 2 samples
or (X_right - X_left) + 1 pels
2X_left + 8
samples
CB
Y
CR
Y
CB
Y
CR
Y
CB
Y
CR
Y
Sample number 2X_left + 9
Sample number 2X_right + 10

Figure 21.21 OSD region horizontal positioning

The top and bottom of the region are defined by the values *Y_top* and *Y_bottom*, which are also in the block header. These values are specified in units of display lines. The top line specified in the first word of an OSD region specification must be greater than or equal to 3.

### 21.4.5 OSD 4:2:2 output

Two, four or eight bits are used to define the color index (i.e. palette address) of each pel. The first, third, fifth, etc. indices are used to reference all three components (Y, $C_B$ and $C_R$) of the respective pels; the second, fourth, sixth, etc. reference only the luminance components of the palette. For this reason the bitmap for a region must define horizontal segments containing a whole number of pel-pairs.

It is possible, however, to define a value of $X_{right}$ such that an odd number of pels will be output in a segment. In this case the index of the bitmap defining the end of each line segment is redundant. Also, at the transition between OSD and the picture, the $C_R$ chrominance value associated with the first pel of the decoded picture display will be defined by the OSD bitmap, not by the picture.

For the same reason, the transition to and from transparency must only occur at points which are an even number of pels from the start of the left-hand edge of an OSD region.

### 21.4.6 Color palette

Each specification block after the first can either define a new palette or use the same palette as the preceding region. If a new palette is defined then it is held in SDRAM immediately after the header and before the bit-map. The P flag in the header defines whether the palette follows the header, as shown in Table 21.4.

| P | Palette |
|---|---|
| 0 | The palette for the region is immediately after header. |
| 1 | The palette is the same as for the preceding region. |

Table 21.4 Palette as before flag

#### Palette modes

The palette mode defines the bits per pel in the bit-map and the pixel resolution. The palette mode can be different for each OSD region, and is defined by the M, Q and E flags in the OSD region specification header. Q defines the pixel resolution, allowing half resolution modes to save memory while retaining the color resolution. The meaning of each combination of these flags is given in Table 21.5.

| M | Q | E | Bits per pixel | No. of colors | Resolution |
|---|---|---|---|---|---|
| 0 | 0 | 0 | 2 | 4 | 1 pel |
| 1 | 1 | 0 | 2 | 4 | 2 pels |
| 1 | 0 | 0 | 4 | 16 | 1 pel |
| 0 | 1 | 0 | 4 | 16 | 2 pels |
| 0 | 0 | 1 | 8 | 256 | 1 pel |
| 1 | 1 | 1 | 8 | 256 | 2 pels |
| 1 | 0 | 1 | Reserved | | |
| 0 | 1 | 1 | Reserved | | |

Table 21.5 M, Q and E palette mode header flags

To reduce the size of the bit maps while retaining the color resolution, a half resolution mode is provided, as shown in Table 21.5. In half resolution, each pel in the bit-map defines the color of two adjacent pixels in the same line in the display.

#### Palette format

The format of each line of the palette defines the format of the output from the palette. Table 21.6 shows the format of the palette lines.

| Field | Bits | Description |
|---|---|---|
| **Cr[3:0]** | 3:0 | Cr chroma value |
| **Cb[3:0]** | 7:4 | Cb chroma value |
| **T** | 8 | Transparency:<br>0 Do NOT blend video with OSD for this color.<br>1 Blend video with OSD for this color using the mix weight $\alpha$2. |
| Reserved | 9 | Reserved. Write 0. |
| **Y[5:0]** | 15:10 | Y luma value |

Table 21.6 Palette line format

### Standard colors

Table 21.7 shows the 14-bit Y, $C_B$ and $C_R$ values nearest to the standard color bar colors.

| Standard color | Y | $C_B$ | $C_R$ |
|---|---|---|---|
| White | 0x3B | 0x8 | 0x8 |
| Black | 0x04 | 0x8 | 0x8 |
| Red | 0x14 | 0x6 | 0xF |
| Green | 0x23 | 0x3 | 0x2 |
| Blue | 0x0C | 0xF | 0x7 |
| Yellow | 0x33 | 0x1 | 0x9 |
| Cyan | 0x2B | 0xA | 0x1 |
| Magenta | 0x1C | 0xD | 0xE |

Table 21.7 Standard colors in 14-bit color

## 21.4.7 OSD bit-map

The bit-map for an OSD region follows the palette if defined or the header if no palette is defined.

The bit-map defines the OSD pixels in left to right order within lines, and the lines in top to bottom order. The number of bits per pixel may be 2, 4 or 8 depending on the palette mode. The value for each pixel gives the line of the palette which defines the color for the pixel.

In 4:2:2 format, the output consists of pairs of pels, where both pels of a pair are needed to define the color. A boundary between the visible video display and a visible portion of OSD should normally be after even-numbered pels, or the next pixel will not be of the correct color. To avoid this effect, an OSD region should start on an even-numbered pel and the width should be an even number of pels. Similarly, any transparent area in the region should start and finish an even number of pels from the edge of the region. This is illustrated in Figure 21.22, where all the marked widths should be an even number of pixels.

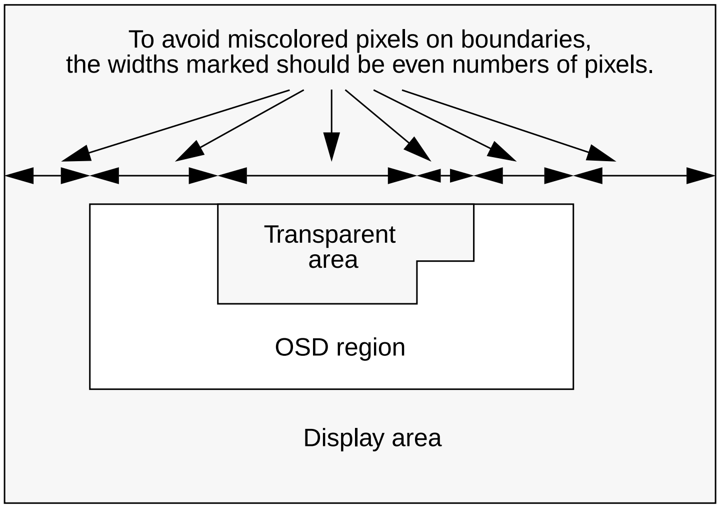

To avoid miscolored pixels on boundaries,
the widths marked should be even numbers of pixels.
Transparent area
OSD region
Display area

Figure 21.22 Avoiding miscolored pixels in 4:2:2 format

### 21.4.8 OSD block header format

Table 21.8 shows the layout of the header, which occupies one 64-bit word. Table 21.9 shows the layout in graphical form, with each line representing a quarter of a 64-bit word.

| Field | Size | Bits | Meaning | Reference |
|---|---|---|---|---|
| M | 1 | 63 | Palette mode. | Table 21.5. |
| Q | 1 | 62 | | |
| E | 1 | 61 | | |
| OSDp[3] | 1 | 60 | Pointer to the next region specification. | Below. |
| P | 1 | 59 | 0 A new palette follows the header.<br>1 The palette is the same as the previous region. | Table 21.4. |
| Y_top | 9 | 56:48 | Position of the top of the OSD region. | Section 21.4.4. |
| MixWeight | 4 | 47:44 | Mixing weight α2 with planes behind. | |
| OSDp[6:4] | 3 | 43:41 | Pointer to the next region specification. | Below. |
| Y_bottom | 9 | 40:32 | Position of the bottom of the OSD region. | Section 21.4.4. |
| OSDp[12:7] | 6 | 31:26 | Pointer to the next region specification. | Below. |
| X_left | 10 | 25:16 | Position of the left of the OSD region. | Section 21.4.4. |
| OSDp[18:13] | 6 | 15:10 | Pointer to the next region specification. | Below. |
| X_right | 10 | 9:0 | Position of the right of the OSD region. | Section 21.4.4. |

Table 21.8 OSD block header format

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| M | Q | E | OSDp<br>[3] | P | 0 | 0 | Y_top |
| MixWeight | | | | OSDp[6:4] | | | Y_bottom |
| OSDp[12:7] | | | | | | X_left | |
| OSDp[18:13] | | | | | | X_right | |

Table 21.9 OSD region specification header

The header contains the pointer OSDp[18:0]. This pointer defines the address of the next block in the linked list to load from memory, as described in section 21.4.3. The blocks can be anywhere in SDRAM and the pointer is given in units of 64-bit words. The block must be 8-word aligned, so the pointer OSDp[18:0] must be a multiple of 8. Thus the least significant 3 bits of OSDp are always zero, and are not included in the header.

The location of the first OSD specification block of a field is defined by the **VID_OBP** or **VID_OTP** registers in units of 128 bytes. This means the full address of the first block must be a multiple of 32.

### 21.4.9 OSD specification block examples

This section shows the format for some complete specification blocks.

Table 21.10 shows a specification using 2 bits per pixel in the bit-map with 1 pel resolution and 14-bit color. Only the first 8 pixels of the bit-map are shown. The palette occupies one 64-bit word, and the bit-map occupies one 64-bit word for every 32 pixels.

| **Bits within a 16-bit quarter-word** | | | | | | | | | | | | | | | | **Description** |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
| 0 | 0 | 0 | 1 | P=0 | 0 | 0 | Y_top | | | | | | | | | Word 0 |
| MixWeight | | | | OSDp[6:4] | | | Y_bottom | | | | | | | | | |
| OSDp[12:7] | | | | | | X_left | | | | | | | | | | |
| OSDp[18:13] | | | | | | X_right | | | | | | | | | | |
| Palette0 Y | | | | | | 0 | T0 | Palette0 Cb | | | | Palette0 Cr | | | | Word 1 |
| Palette1 Y | | | | | | 0 | T1 | Palette1 Cb | | | | Palette1 Cr | | | | |
| Palette2 Y | | | | | | 0 | T2 | Palette2 Cb | | | | Palette2 Cr | | | | |
| Palette3 Y | | | | | | 0 | T3 | Palette3 Cb | | | | Palette3 Cr | | | | |
| Bit-map for 8 OSD pixels | | | | | | | | | | | | | | | | Bit-map word |

Table 21.10 2 bits per pixel, 14-bit color OSD region specification

Table 21.11 shows a specification using 4 bits per pixel in the bit-map with 2 pel resolution and 14-bit color. Only the first 4 pixels of the bit-map are shown. Each entry in the bit-map uses 4 bits, but defines two display pels of the same color. The palette occupies four 64-bit words, and the bit-map occupies one 64-bit word for every 16 bit-map pixels.

| Bits within a 16-bit quarter-word | | | | | | | | | | | | | | | | Description |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
| M=1 | Q=1 | E=0 | 1 | P=0 | 0 | 0 | Y_top | | | | | | | | | Word 0 |
| MixWeight | | | | OSDp[6:4] | | | Y_bottom | | | | | | | | | |
| OSDp[12:7] | | | | | | X_left | | | | | | | | | | |
| OSDp[18:13] | | | | | | X_right | | | | | | | | | | |
| Palette0 Y | | | | | | 0 | T0 | Palette0 Cb | | | | Palette0 Cr | | | | Word 1 |
| Palette1 Y | | | | | | 0 | T1 | Palette1 Cb | | | | Palette1 Cr | | | | |
| Palette2 Y | | | | | | 0 | T2 | Palette2 Cb | | | | Palette2 Cr | | | | |
| Palette3 Y | | | | | | 0 | T3 | Palette3 Cb | | | | Palette3 Cr | | | | |
| Palette4 Y | | | | | | 0 | T4 | Palette4 Cb | | | | Palette4 Cr | | | | Word 3 |
| Palette5 Y | | | | | | 0 | T5 | Palette5 Cb | | | | Palette5 Cr | | | | |
| Palette6 Y | | | | | | 0 | T6 | Palette6 Cb | | | | Palette6 Cr | | | | |
| Palette7 Y | | | | | | 0 | T7 | Palette7 Cb | | | | Palette7 Cr | | | | |
| Palette8 Y | | | | | | 0 | T8 | Palette8 Cb | | | | Palette8 Cr | | | | Word 4 |
| Palette9 Y | | | | | | 0 | T9 | Palette9 Cb | | | | Palette9 Cr | | | | |
| Palette10 Y | | | | | | 0 | T10 | Palette10 Cb | | | | Palette10 Cr | | | | |
| Palette11 Y | | | | | | 0 | T11 | Palette11 Cb | | | | Palette11 Cr | | | | |
| Palette12 Y | | | | | | 0 | T12 | Palette12 Cb | | | | Palette12 Cr | | | | Word 5 |
| Palette13 Y | | | | | | 0 | T13 | Palette13 Cb | | | | Palette13 Cr | | | | |
| Palette14 Y | | | | | | 0 | T14 | Palette14 Cb | | | | Palette14 Cr | | | | |
| Palette15 Y | | | | | | 0 | T15 | Palette15 Cb | | | | Palette15 Cr | | | | |
| Bit-map for 4 OSD pixels | | | | | | | | | | | | | | | | Bit-map word |

Table 21.11 4 bits per pixel, 2-pel resolution 14-bit color OSD region specification

Table 21.12 shows a specification using 8 bits per pixel in the bit-map with full resolution and 14-bit color. Only the first 2 pixels of the bit-map are shown. Each pixel in the bit-map uses 8 bits. The palette occupies 64 64-bit words (i.e. 512 bytes), and the bit-map occupies one 64-bit word for every 8 bit-map pixels.

| **Bits within a 16-bit quarter-word** | | | | | | | | | | | | | | | | **Description** |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
| 0 | 0 | 1 | 1 | 0 | 0 | 0 | Y_top | | | | | | | | | Word 0 |
| MixWeight | | | | OSDp[6:4] | | | Y_bottom | | | | | | | | | |
| OSDp[12:7] | | | | | | X_left | | | | | | | | | | |
| OSDp[18:13] | | | | | | X_right | | | | | | | | | | |
| Palette0 Y | | | | | | 0 | T0 | Palette0 Cb | | | | Palette0 Cr | | | | Word 1 |
| Palette1 Y | | | | | | 0 | T1 | Palette1 Cb | | | | Palette1 Cr | | | | |
| Palette2 Y | | | | | | 0 | T2 | Palette2 Cb | | | | Palette2 Cr | | | | |
| Palette3 Y | | | | | | 0 | T3 | Palette3 Cb | | | | Palette3 Cr | | | | |
| Palette4 Y | | | | | | 0 | T4 | Palette4 Cb | | | | Palette4 Cr | | | | Word 2 |
| ... | | | | | | | | ... | | | | ... | | | | ... |
| Palette252 Y | | | | | | 0 | T12 | Palette252 Cb | | | | Palette252 Cr | | | | Word 64 |
| Palette253 Y | | | | | | 0 | T13 | Palette253 Cb | | | | Palette253 Cr | | | | |
| Palette254 Y | | | | | | 0 | T14 | Palette254 Cb | | | | Palette254 Cr | | | | |
| Palette255 Y | | | | | | 0 | T15 | Palette255 Cb | | | | Palette255 Cr | | | | |
| Bit-map for 2 OSD pixels with 8 bits per pel or 8 bits per 2 pel | | | | | | | | | | | | | | | | Bit-map Word |

Table 21.12  8 bits per pixel, 14-bit color OSD region specification

### 21.4.10 Mixing OSD with video

The mixing function allows each OSD pixel to be blended with the corresponding pixel generated by the planes behind the OSD. The mix weight is a programmable parameter and can be set for each OSD region.

The mix weight is a 4-bit number allowing mixing ratios from 0 to 1 with a resolution of 1/15. The resulting pixel can be completely transparent (weighting of 0/15) or can completely cover the video (15/15). Each individual color in the palette can be specified to be used with or without mixing by setting the transparency (T) bit of the palette. A T bit equal to 0 means no mixing for the particular color, and a T of 1 means that mixing should be used.

### 21.4.11OSD active signal

The OSD active signal can be used in two modes. The mode is controlled using **VID_DCF.OAM**. In the first mode the OSD active signal is configured as an output. In this mode the OSD active signal denotes when an active OSD pixel (non transparent) is on the YC output bus, as in Figure 21.23. The signal, in this mode, has a programmable delay controlled by **VID_DCF.OAD**. This delay can be set such that the OSD active signal is set as much as two clocks before or 1 to 64 clocks after the actual pixel.

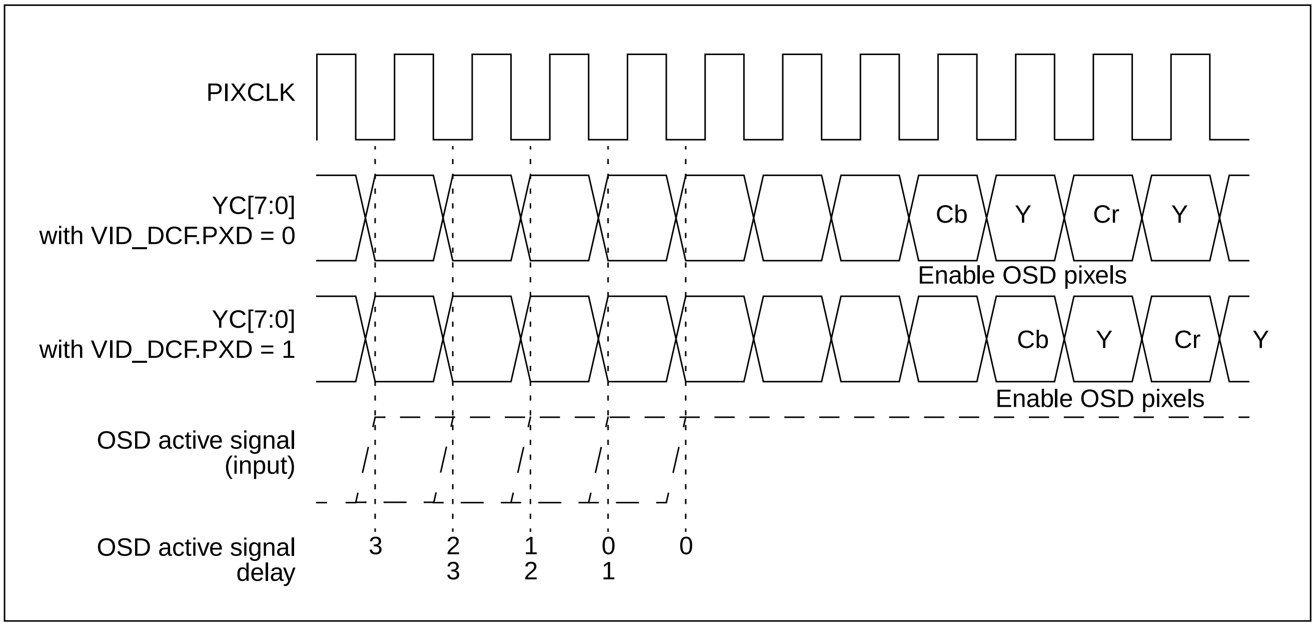


Figure 21.23 OSD active timing when **VID_DCF.OAM** = 1

In the second mode of operation, the OSD active signal is configured as an input and is used to disable the OSD. When the signal goes high, the OSD will be placed on the YC bus if the OSD is enabled. When this signal is low then no OSD will be placed on the bus even if OSD is enabled. The programmable delay is used in the same way as for the input signal.

| OSD active mode | OSD active signal | Meaning |
|---|---|---|
| 0 | 0 | Signal is an output. Video Pixels only on display bus. |
| 0 | 1 | Signal is an output. OSD Pixels on the display bus. |
| 1 | 0 | Signal is an input. Disable the OSD output. |
| 1 | 1 | Signal is an input. Enable OSD output if available. |

Table 21.13 OSD active signal operation

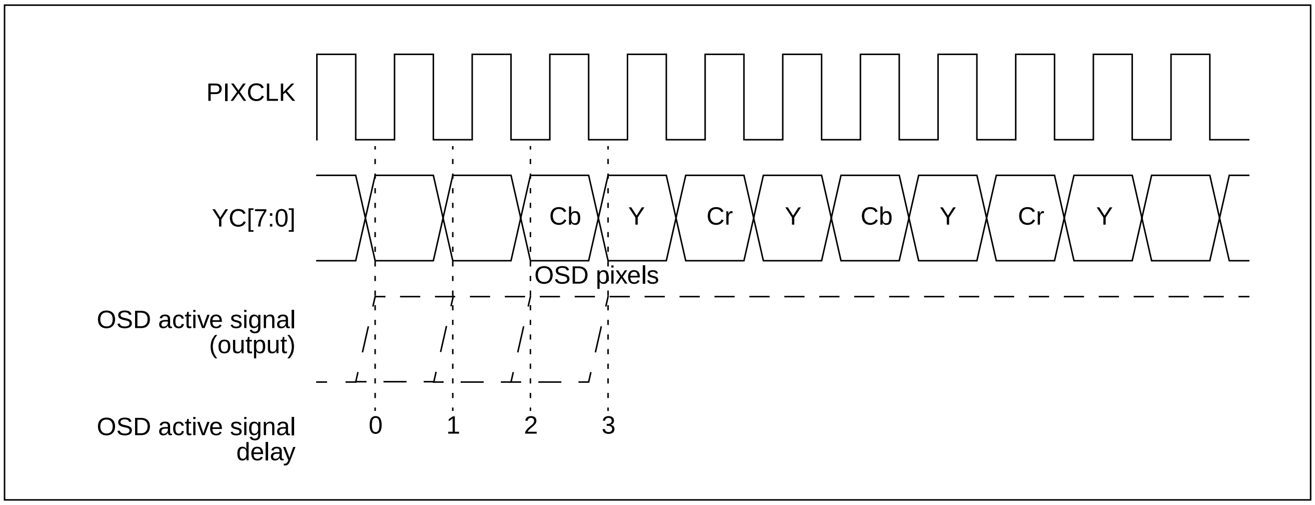


Figure 21.24 OSD active timing when **VID_DCF.OAM** = 0

## 21.5 Mixing display planes

The blending of the elements of the final picture is performed by a mixing unit, which is shown in Figure 21.25. The mixing of the display planes is controlled by the mix weights, α1 and α2.

The mix weight α1 controls the mixing of the back two planes, the sub-picture plane and the video plane. α1 comes from the sub-picture stream.

The result of mixing the back two planes can be blended in turn with the OSD. The OSD mix weight is α2 and is defined in each palette. It is a 4-bit value, defined for each OSD region, and mixing can be enabled or disabled for each color.

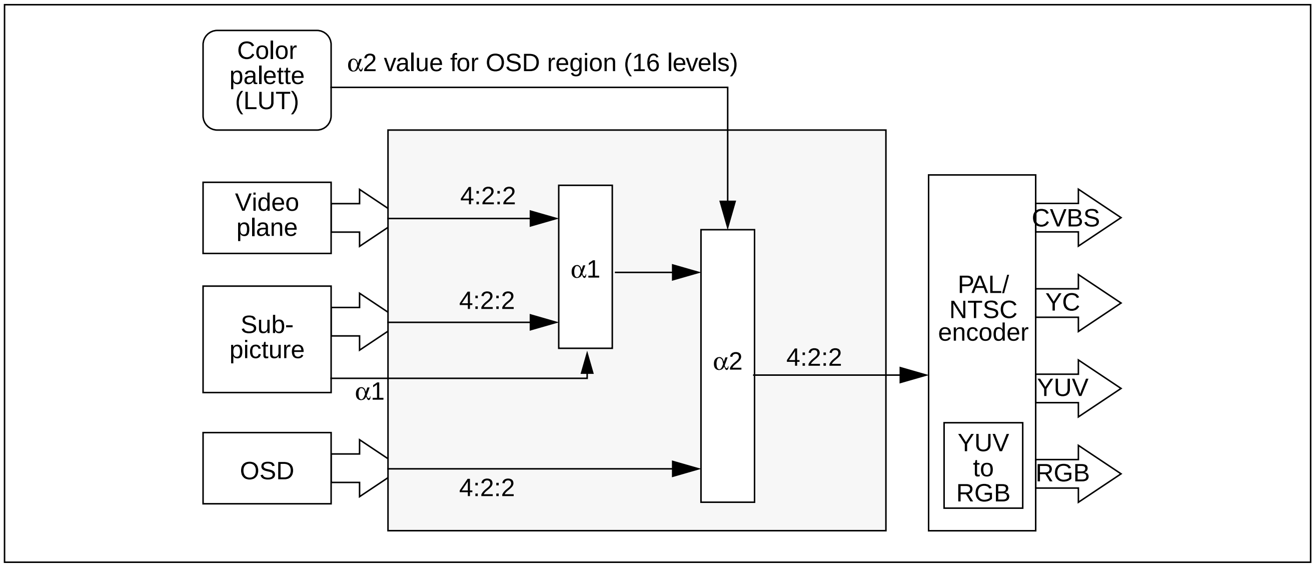


Figure 21.25 Mixing unit

# 22 Teletext interface

The STi5500 has a teletext interface (**TtxtInt**) which interfaces to a teletext peripheral. It translates teletext data from memory. It has a single mode of operation, Telextext data out.

The teletext interface uses DMA to retrieve teletext data from memory, and serializes the data for transmission to a composite video encoder.

The interface between the CPU and the teletext interface is not a channel model but is based on an interrupt mechanism.

## 22.1 Teletext interface internal signals

| Pin | In/Out | Function |
|---|---|---|
| **TtxtData** | out | Teletext serial data |
| **TtxtEvennotOdd** | in | Teletext even not odd |
| **TtxtRequest** | in | Teletext serial data request input. |
| **TtxtClock** | in | 27 MHz teletext clock |

Table 22.1 Teletext interface pins

## 22.2 Teletext data out

The teletext interface uses DMA to retrieve teletext data from memory, and serializes the data for transmission to a composite video encoder. Clock run-in bits are added to the start of the serial stream, as defined in the ETSI specification[1].

The CPU is responsible for assuring the correct programming of the video encoder. The encoder must be programmed such that it makes requests for teletext lines only on pre-specified lines.

The **TtxtEvennotOdd** input from the encoder is used to interrupt the CPU allowing software control of the teletext out DMA initialisation.

The CPU initiates the output of a number of lines of teletext data. These lines are output when suitable requests are made from the video encoder. The teletext interface uses the device protocols to allow control by the CPU.

### 22.2.1 Format of the output line

One teletext line is output as a stream of 360 bits, at an average frequency of 6.9375 MHz. The line is composed of two bytes of clock run-in (16 bits), followed by the data extracted from the transport packet. The data field consists of the framing_code, magazine_and_packet_address, and data_block fields. These three fields provide the block of teletext data.

The clock run-in is composed of two bytes of ‘10101010’. The framing code, which is extracted from the data_field, should be a single byte of ‘11100100’[2]. Hence one line of teletext output will be

[1] Specification for conveying ITU-R Systems B Teletext in Digital Video Broadcasting (DVB) bitstreams.

[2] Document SPB492, ‘Teletext Specification’. European Broadcasting Union, Geneva, December 1992.

composed as in Figure 22.1. The data will be transmitted from least significant bit (LSB) to most significant bit (MSB) of each byte in memory.

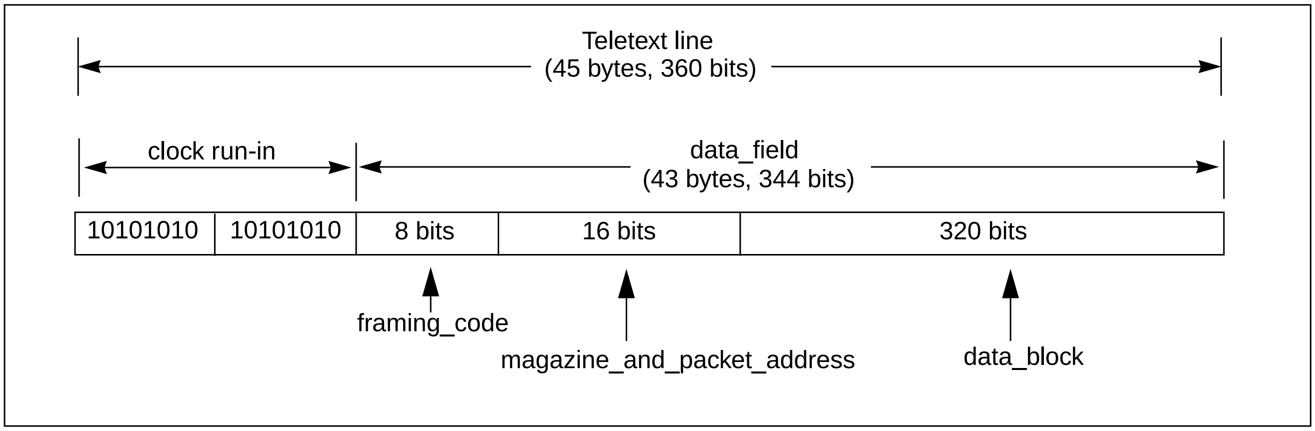


Figure 22.1 Line output

The 360 bits of output data are defined to be nine 37-bit sequences, ending with one 27-bit sequence. Within each sequence, all bits are transmitted using four 27 MHz cycles, except bits 10, 19, 28 and 37, which are transmitted using three 27 MHz cycles, see Figure 22.2.

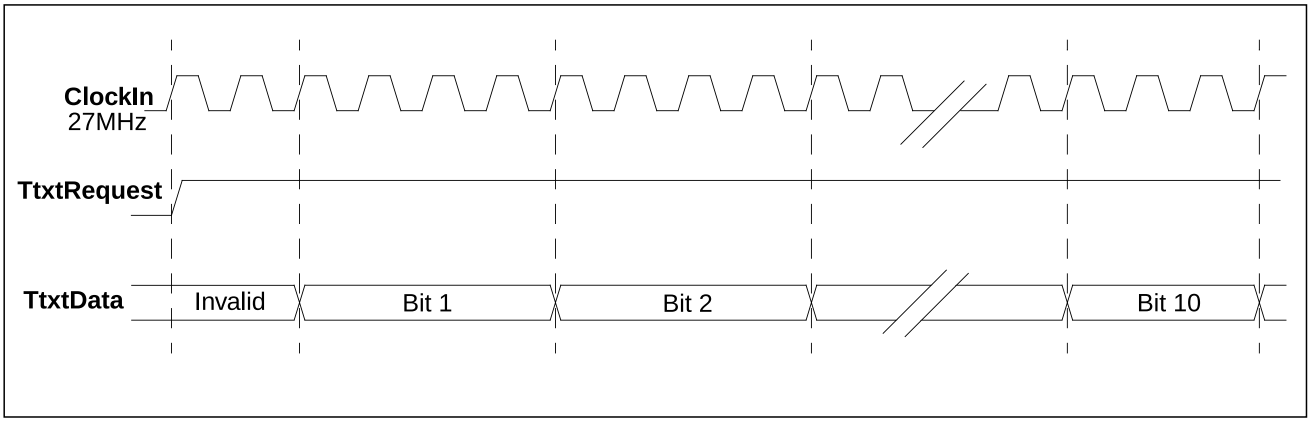


Figure 22.2 Output data

## 22.3 Teletext interrupt control

The teletext interface can be programmed, via the **TtxtIntEnable** register, to interrupt the CPU whenever one of the following occurs:

- a teletext data out data transfer completes
- the current video frame toggles odd to even or even to odd

The interrupt status contained within the **TtxtIntStatus** register is masked with the **TtxtIntEnable** register. The interrupt bits are reset when the CPU writes to the specific acknowledgement register, or when a DMA operation completes.

## 22.4 Control registers

The teletext interface is programmable via configuration registers.

### TtxtDmaAddress register

The **TtxtDmaAddress** register is a 32-bit read/write register. It specifies the DMA start location of data from memory.

| **TtxtDmaAddress** | **Ttxt base address + #00** | **Read/Write** |
|---|---|---|
| **Bit** | **Bit field** | **Function** |
| 31:0 | **DmaAddress** | DMA start location of data from memory. |

Table 22.2 **TtxtDmaAddress** register format

### TtxtDmaCount register

The **TtxtDmaCount** register specifies the number of bytes to be transferred from memory during the DMA operation. This value must be a multiple (*n*) of 46 bytes, where *n* is the number of lines to output.

A write to this register also arms the teletext out operation.

| **TtxtDmaCount** | **Ttxt base address + #04** | **Read/Write** |
|---|---|---|
| **Bit** | **Bit field** | **Function** |
| 10:0 | **DMACount** | Specifies the number of bytes to be transferred during the DMA operation from memory and starts the DMA. |

Table 22.3 **TtxtDmaCount** register format

### TtxtOutDelay register

This register is used to program the delay, in 27 MHz clock periods, from **TtxtRequest** to **TtxtData**.

| **TtxtOutDelay** | **Ttxt base address + #08** | **Read/Write** |
|---|---|---|
| **Bit** | **Bit field** | **Function** |
| 8:0 | **Delay** | Delay from the rising edge of **TtxtRequest** to the first valid teletext data bit in 27MHz clock periods. Valid values are in the range 2 to 9. |

Table 22.4 **TtxtOutDelay** register format

### TtxtMode register

This register sets the mode of the teletext interface. It specifies whether teletext data in memory is for odd or even fields.

| **TtxtMode** | **Ttxt base address + #14** | **Read/Write** |
|---|---|---|
| **Bit** | **Bit field** | **Function** |
| 1 | **OddEven** | Specifies odd or even fields of teletext data.<br>0 Teletext data to/from memory is for EVEN fields<br>1 Teletext data to/from memory is for ODD fields |

Table 22.5 **TtxtMode** register format

### TtxtIntStatus register

This register gives the current state of the teletext interface operations.

| TtxtIntStatus | Ttxt base address + #18 | Read |
|---|---|---|
| **Bit** | **Bit field** | **Function** |
| 0 | **OutComplete** | Teletext out operation completed. |
| 1 | **Odd** | Current (video encoder) field is ODD. |
| 2 | **Even** | Current (video encoder) field is EVEN. |

Table 22.6 **TtxtIntStatus** register format

### TtxtIntEnable register

This register allows masking of the **TtxtIntStatus** register.

| TtxtInttEnable | Ttxt base address + #1C | Read/Write |
|---|---|---|
| **Bit** | **Bit field** | **Function** |
| 0 | **InOutCompleteEn** | Enable teletext out operation completed interrupt. |
| 1 | **OddEnable** | Enable odd field interrupt. |
| 2 | **EvenEnable** | Enable even field interrupt. |

Table 22.7 **TtxtIntEnable** register format

### TtxtAckOddEven register

This register is address sensitive only and clears the **Odd** and **Even** bits of the **TtxtIntStatus** register.

| TtxtAckOddEven | Ttxt base address + #20 | Write |
|---|---|---|
| **Bit** | **Bit field** | **Function** |
|  | **AckOddEven** | Acknowledge odd/even toggle interrupt. |

Table 22.8 **TtxtAckOddEven** register format

### TtxtAbort register

This register is write only and address sensitive only. A write to this address causes the teletext interface to abort the current operation. The state of the teletext out operation is reset, and the teletext data transfer is interrupted. The DMA engine is reset only after the current word read/write is complete.

| TtxtAbort | Ttxt base address + #24 | Write only |
|---|---|---|
| **Bit** | **Bit field** | **Function** |
|  | **Abort** | Abort current operation. |

Table 22.9 **TtxtAbort** register format

# 23 PAL/NTSC encoder (DENC)

## 23.1 Description

The STi5500 contains a high performance PAL/NTSC digital encoder, sometimes referred to as the DENC. The encoder converts the 4:2:2 or 4:4:4 digital video stream and the OSD, sub-picture and picture planes into a standard analog base-band PAL/NTSC signal and into RGB analog components. The encoder outputs interlaced or non-interlaced video in PAL-B, D, G, H, I, PAL-N, PAL-M, NTSC-M or NTSC- 4.43 standards.

Six analog output pins are available on which it is possible to output CVBS, S-VHS (Y/C), RGB and YUV formats. The encoder can handle interlaced mode (with 525- or 625-line standards) and non-interlaced mode. It can perform closed-captions, CGMS or Teletext encoding.

The encoder can operate either in master mode, where it supplies all sync signals, or in one of several slave modes, where it locks onto incoming sync signals. An autotest mode is also provided.

The main functions are controlled using 8-bit configuration registers accessed by software. See section 23.20 for a list of the configuration registers available.

## 23.2 Video timing

The burst sequences are internally generated, subcarrier generation being performed numerically with CKREF as reference. 4-frame bursts are generated for PAL or 2-frame bursts for NTSC. Rise and fall times of synchronization tips and burst envelope are internally controlled according to the relevant ITU-R and SMPTE recommendations. CKREF is the 27 MHz input clock to the STi5500

Figures 23.2, 23.3, 23.4, 23.5, 23.6 and 23.7 depict typical VBI waveforms.

It is possible to allow encoding of incoming YCrCb data on those lines of the VBI that do not bear line sync pulses or pre/post-equalization pulses (see Figures 23.2, 23.3, 23.4, 23.5, 23.6 and 23.7). This mode of operation is referred to as *partial blanking* and is the default set-up. It allows the encoded waveform to keep any VBI data present in digitized form in the incoming YCrCb stream (e.g. supplementary Closed-Captions line or StarSight data). Alternatively, the complete VBI may be *fully blanked*, so no incoming YCrCb data is encoded on these lines.

Full or partial blanking is controlled by bit **blkli** in register **configuration1**.

For 525/60 systems, with the SMPTE line numbering convention:

- the complete VBI consists of lines 1 to 19 and the second half of lines 263 to 282;
- the partial VBI consists of lines 1 to 9 and the second half of lines 263 to 272;
- line 282 is either fully blanked or fully active.

For 625/50 systems, with the CCIR line numbering convention:

- the complete VBI consists of the second half of lines 623 to 22 and lines 311 to 335;
- the partial VBI consists of the second half of lines 623 to 5 and lines 311 to 318;
- line 23 is always fully active.

In an ITU-R656-compliant digital TV line, the active portion of the digital line is the portion included between the SAV (Start of Active Video) and EAV (End of Active Video) words. However, this digital active line starts somewhat earlier and may end slightly later than the active line usually defined by analog standards. The DENC allows two approaches:

- It is possible to encode the full digital line (720 pixels / 1440 clock cycles). In this case, the output waveform will reflect the full YCrCb stream included between SAV and EAV.
- Alternatively, it is possible to drop some YCrCb samples at the extremities of the digital line so that the encoded analog line fits within the analog ITU-R/SMPTE specifications.

Selection between these two modes of operation is performed with bit **aline** in register **configuration4**.

In all cases, the transitions between horizontal blanking and active video are shaped to avoid too steep edges within the active video. Figure 23.8 and Table 23.1 give typical timings concerning the horizontal blanking interval and the active video interval. Actual values will depend on the static offset programmed for subcarrier generation.

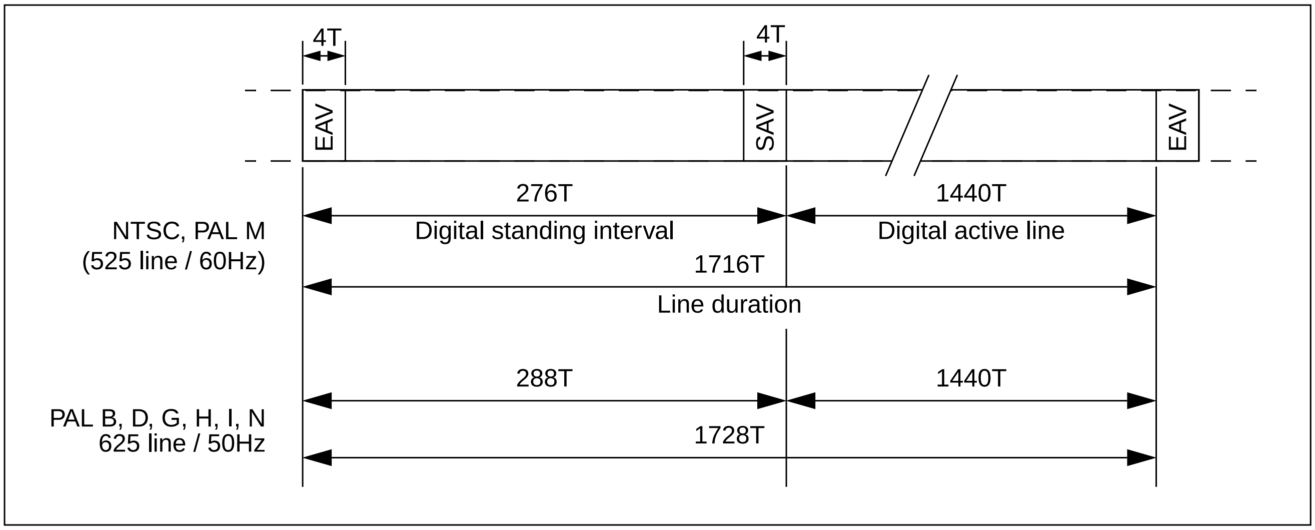
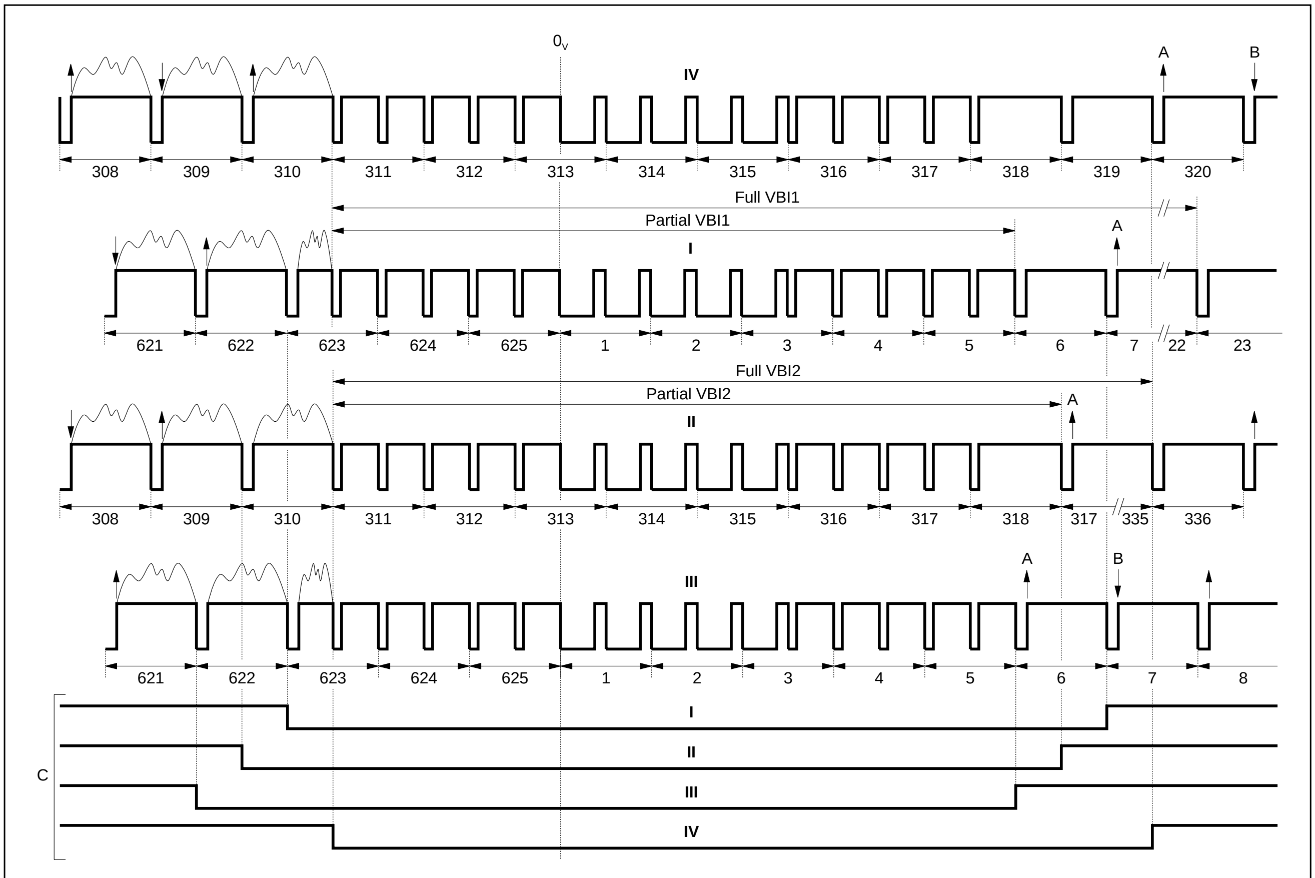


Figure 23.1 Input data format



0v : Frame synchronization reference
I, II, III, IV : 1st and 5th, 2nd and 6th, 3rd and 7th, 4th and 8th fields
A : Burst phase : nominal value +135°
B : Burst phase : nominal value -135°
C : Burst suppression internal

Figure 23.2 PAL-BDGHI, PAL-N typical VBI waveform, interlaced mode (ITU-R625 line numbering)

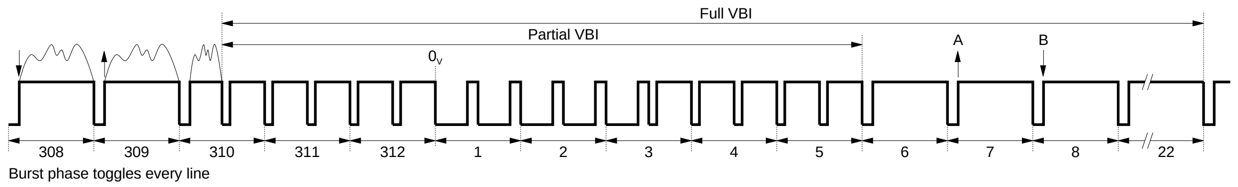


Figure 23.3 PAL-BDGHI, PAL-N typical VBI waveform, non-interlaced mode (“CCIR-like” line numbering

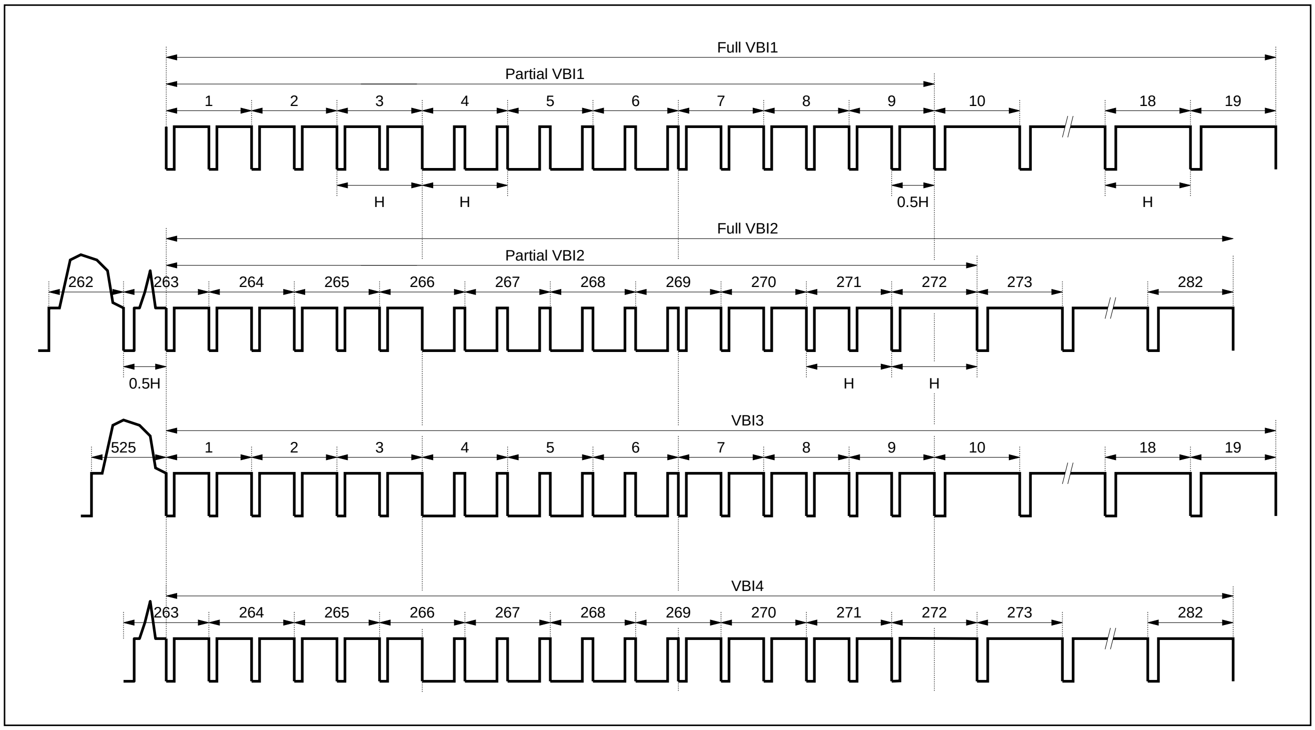


Figure 23.4 NTSC-M typical VBI waveforms, interlaced mode (SMPTE-525 line numbering)

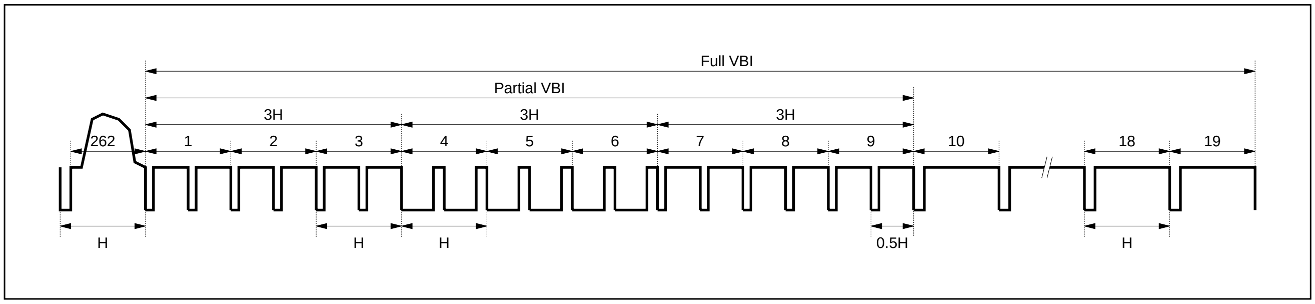


Figure 23.5 NTSC-M typical VBI waveforms, non-interlaced mode (SMPTE-like line numbering)

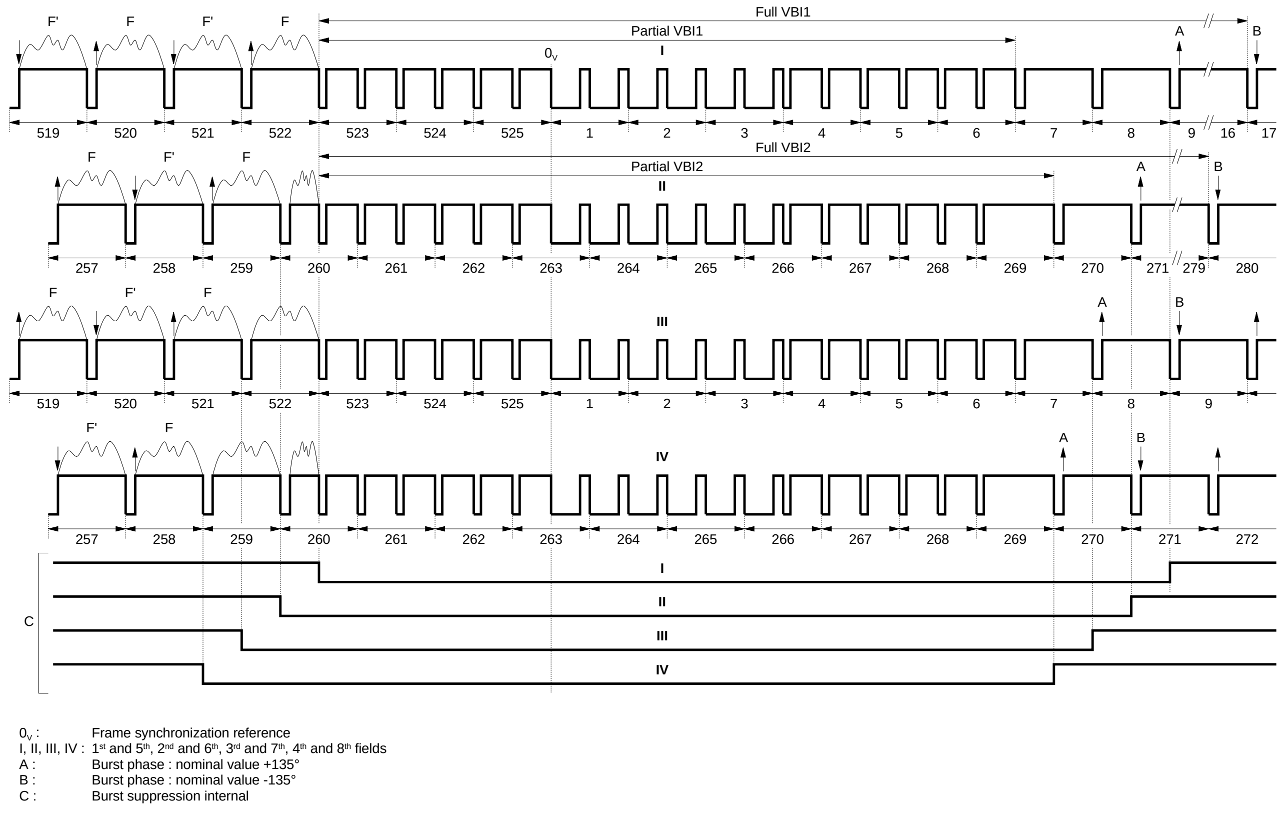


Figure 23.6 PAL-M typical VBI waveforms, interlaced mode (ITU-R/CCIR-525 line numbering)

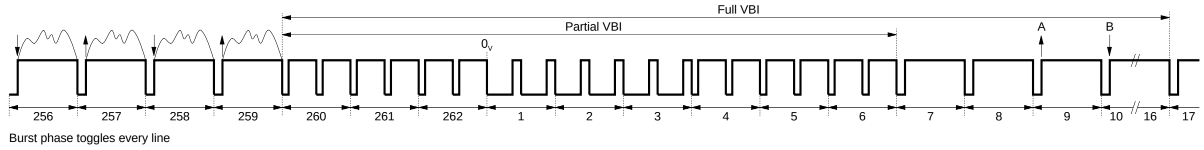


Figure 23.7 PAL-M typical VBI waveforms, non-interlaced mode (ITU-R/CCIR-like line numbering)

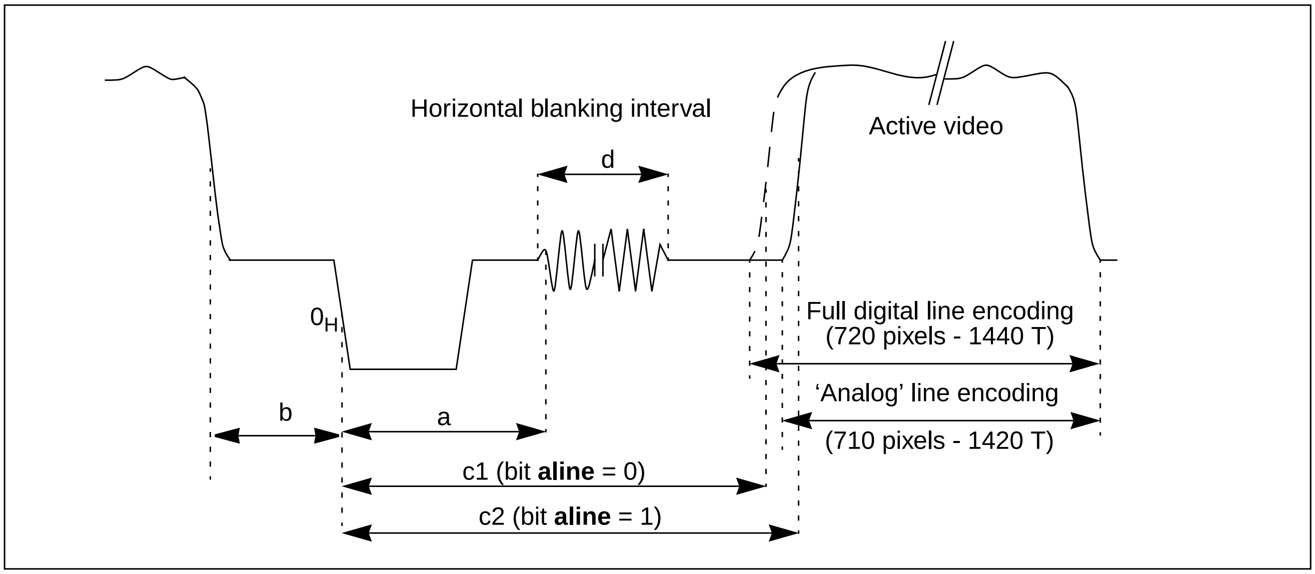


Figure 23.8 Horizontal blanking interval and active video timings

| | NTSC-M | PAL-BDGHI | PAL-N | PAL-M |
|---|---|---|---|---|
| a | 5.38 µs (even lines)<br>5.52 µs (odd lines) | 5.54 µs (A-type)<br>5.66 µs (B-type) | 5.54 µs (A-type)<br>5.66 µs (B-type) | 5.73 µs (A-type)<br>5.87 µs (B-type) |
| b | 1.56 µs | 1.28 µs | 1.28 µs | 1.28 µs |
| c1 | 8.8 µs | 9.3 µs | 9.3 µs | 9.3 µs |
| c2 | 9.3 µs | 10.1 µs | 10.1 µs | 10.1 µs |
| d | 9 cycles of 3.58MHz | 10 cycles of 4.43MHz | 9 cycles of 3.58MHz | 9 cycles of 3.58MHz |

Table 23.1 Typical timing values in Figure 23.8

## 23.3 Reset procedure

A hardware reset is performed by grounding the pin NRESET. The master clock must be running and pin NRESET kept low for a minimum of 5 clock cycles. This sets the DENC in HSYNC+ODDEVEN (line-locked) slave mode, for NTSC-M, interlaced ITU-R601 encoding with Macrovision$^{TM}$ copy protection revision 7.01 operating. Closed-captioning and Teletext encoding are all disabled.

Then the configuration can be customized by writing into the appropriate registers. A few registers are never reset, their contents are unknown until the first loading (see the STi5500 Register Manual).

It is also possible to perform a software reset by setting the bit **softreset** in the register **configuration6**. The response of the device in that case is similar to its response after a hardware reset, except that the configuration registers and a few other registers are not altered. For further details see the description of bit **softreset**.

## 23.4 Master mode

In this mode, the DENC supplies HSYNC and ODDEVEN sync signals (with independently programmable polarities) to drive other blocks. Refer to Figure 23.9 and Figure 23.10 for timings and waveforms.

The DENC starts encoding and counting clock cycles as soon as the master mode has been loaded into the **configuration0** register.

Bits **syncout_ad[1:0]** of register **configuration4** allow software to shift the relative position of the sync signals by up to 3 clock cycles to cope with any YCrCb phasing.

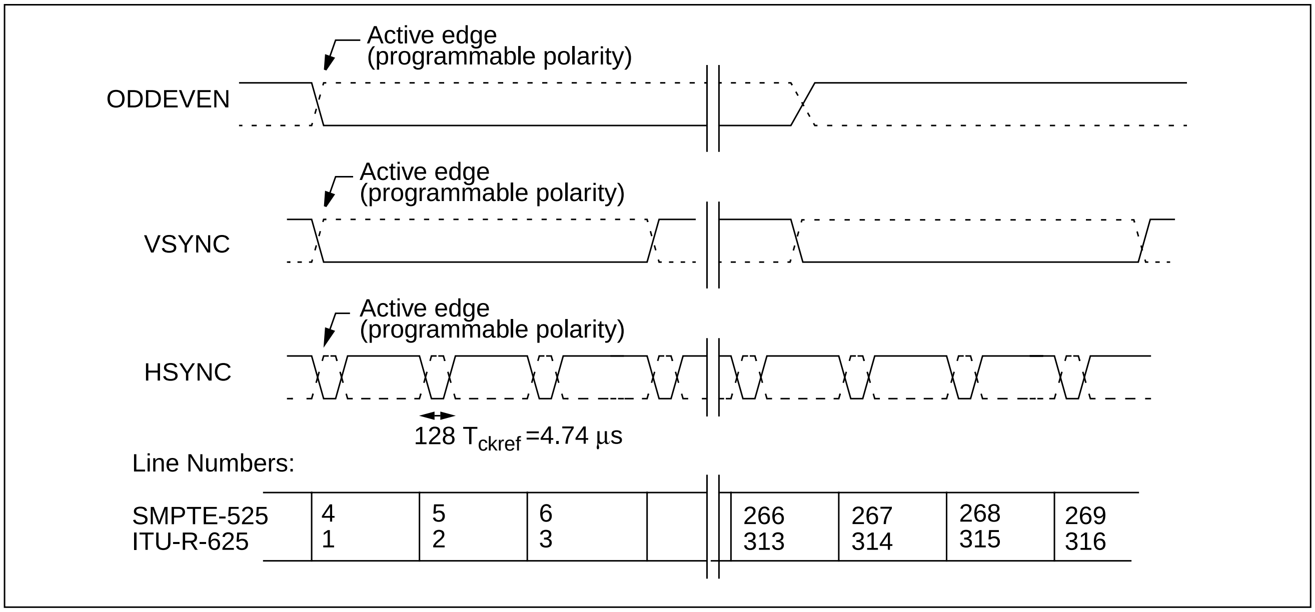

Active edge (programmable polarity)
ODDEVEN
Active edge (programmable polarity)
VSYNC
Active edge (programmable polarity)
HSYNC
128 $T_{ckref}$ =4.74 µs
Line Numbers:
SMPTE-525
ITU-R-625
4
1
5
2
6
3
266
313
267
314
268
315
269
316

Figure 23.9 ODDEVEN, VSYNC and HSYNC waveforms

Note 1: When ODDEVEN is a sync input, only one edge, the *active* edge, of the incoming ODDEVEN is taken into account for synchronization. The *inactive* edge (the second edge on this drawing) is not critical and its position may differ by H/2 from the location shown.

Note 2: The HSYNC pulse width indicated is valid when the DENC supplies HSYNC. In those slave modes where it receives HSYNC, only the edge defined as active is relevant, and the width of the HSYNC pulse it receives is not critical.

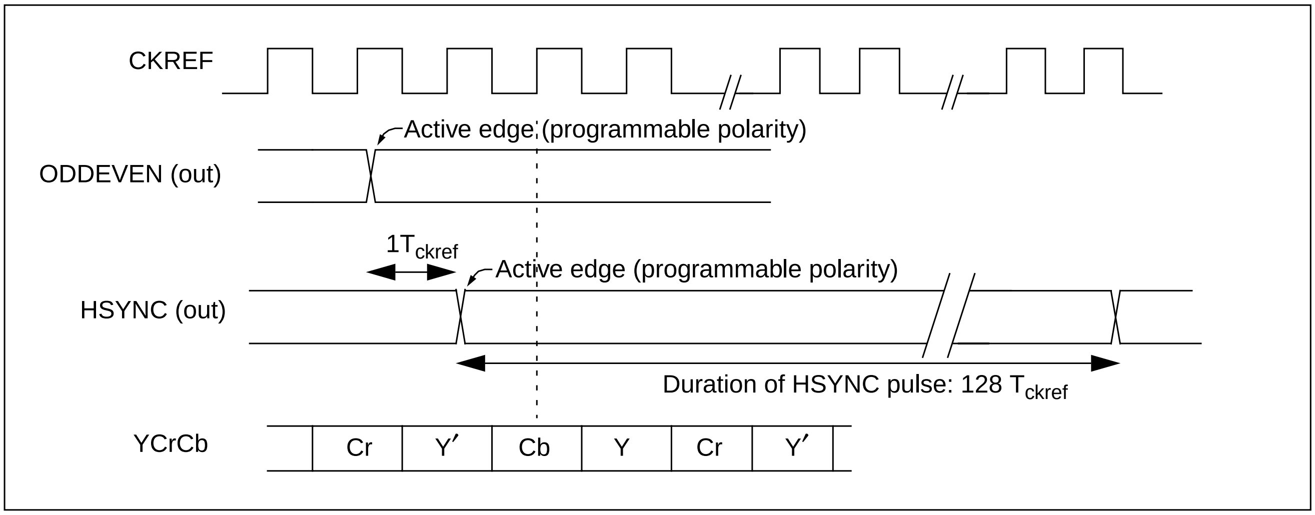

CKREF
ODDEVEN (out)
Active edge (programmable polarity)
$1T_{ckref}$
Active edge (programmable polarity)
HSYNC (out)
Duration of HSYNC pulse: 128 $T_{ckref}$
YCrCb
Cr
Y′
Cb
Y
Cr
Y′

Figure 23.10 Master mode sync signals

Note: This figure is valid for bits **syncout_ad[1:0]** = default

## 23.5 Slave modes

Several slave modes are available:

- HSYNC + ODDEVEN based (line-based sync),
- HSYNC + VSYNC based (another type of line-based sync),
- ODDEVEN-only based (frame-based sync),
- VSYNC-only based (another type of frame-based sync),
- sync-in-data based (line locked or frame locked).

ODDEVEN refers to an odd/even field flag, also known as BottomnotTop. HSYNC is a line sync signal, VSYNC is a vertical sync signal. The waveforms of these signals are depicted in Figure 23.9. The polarities of HSYNC and VSYNC/ODDEVEN are independently programmable in all slave modes.

In all slave modes, ODDEVEN (VSYNC) and/or HSYNC signals must be related to **Pixclk**, the principal DENC clock. In other words, there is *no genlocking* performed by the DENC.

### 23.5.1 Synchronization onto a line sync signal

**HSYNC+ODDEVEN based synchronization**

Synchronization is performed on a line-by-line basis by locking onto incoming ODDEVEN and HSYNC signals. See Figure 23.11 for waveforms and timings. The polarities of the active edges of HSYNC and ODDEVEN are programmable and independent.

The first active edge of ODDEVEN initializes the internal line counter but encoding of the first line does not start until an HSYNC active edge is detected (at the earliest, an HSYNC transition may be at the same time as ODDEVEN). At that point, the internal sample counter is initialized and encoding of the first line starts. Then, encoding of each subsequent line is individually triggered by HSYNC active edges. The phase relationship between HSYNC and the incoming YCrCB data is normally such that the first clock rising edge following the HSYNC active edge samples Cb (i.e. a blue chroma sample within the YCrCb stream). It is however possible to internally delay the incoming sync signals (HSYNC+ODDEVEN) by up to 3 clock cycles to cope with different data/sync phasings, using configuration bits **syncin_ad** in **configuration4**.

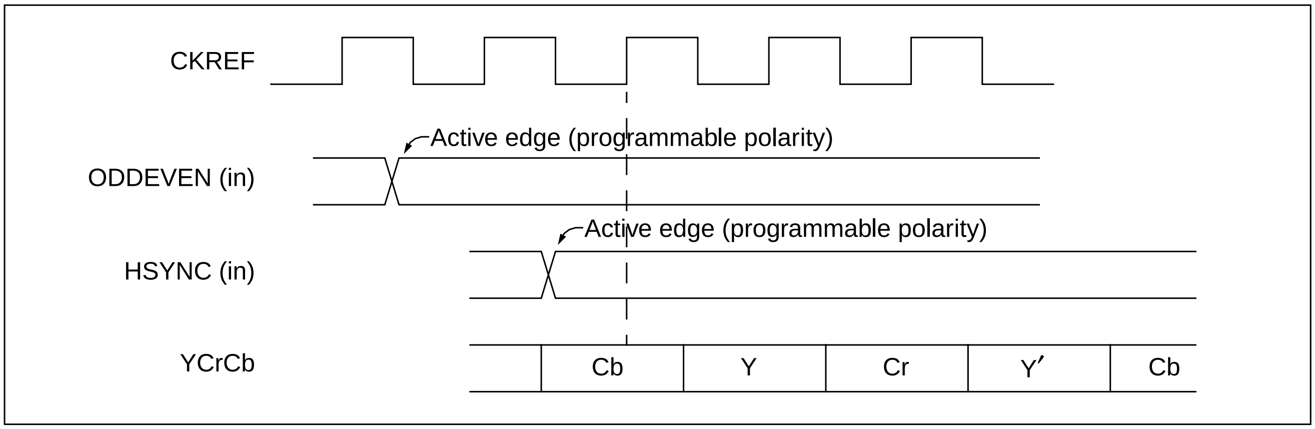

CKREF
Active edge (programmable polarity)
ODDEVEN (in)
Active edge (programmable polarity)
HSYNC (in)
YCrCb
Cb
Y
Cr
Y′
Cb

Figure 23.11 HSYNC + ODDEVEN based slave mode sync signals

Note: This figure is valid for bits **syncin_ad[1:0]** = default

The DENC is thus fully slaved to the HSYNC signal, which means that lines may contain more or less samples than usual.

- If the digital line is shorter than its nominal value, the sample counter is re-initialized when the ‘early’ HSYNC arrives and all internal synchronization signals are re-initialized.
- If the digital line is longer than its nominal value, the sample counter is stopped when it reaches its nominal end-of-line value and waits for the ‘late’ HSYNC before reinitializing.

The field counter is incremented on each ODDEVEN transition. The line counter is reset on the HSYNC following each active edge of ODDEVEN.

### HSYNC+VSYNC based synchronization

Synchronization is performed on a line-by-line basis by locking onto incoming VSYNC and HSYNC signals. Refer to Figure 23.12 for waveforms and timings. The polarities of HSYNC and VSYNC are programmable and independent.

The incoming VSYNC signal is immediately transformed into a waveform identical to the odd/even waveform of an ODDEVEN signal, therefore the behaviour of the core is identical to that described above for HSYNC+ODDEVEN based synchronization. Again, the phase relationship between HSYNC and the incoming YCrCb data is normally such that the first clock rising edge following the HSYNC active edge samples “Cb” (i.e. a ‘blue’ chroma sample within the YCrCb stream). It is however possible to internally delay the incoming sync signals (HSYNCand VSYNC) by up to 3 clock cycles to cope with different data/sync phasings, using configuration bits **Syncin_ad** of **configuration4**.

The field counter is incremented on each active edge of VSYNC.

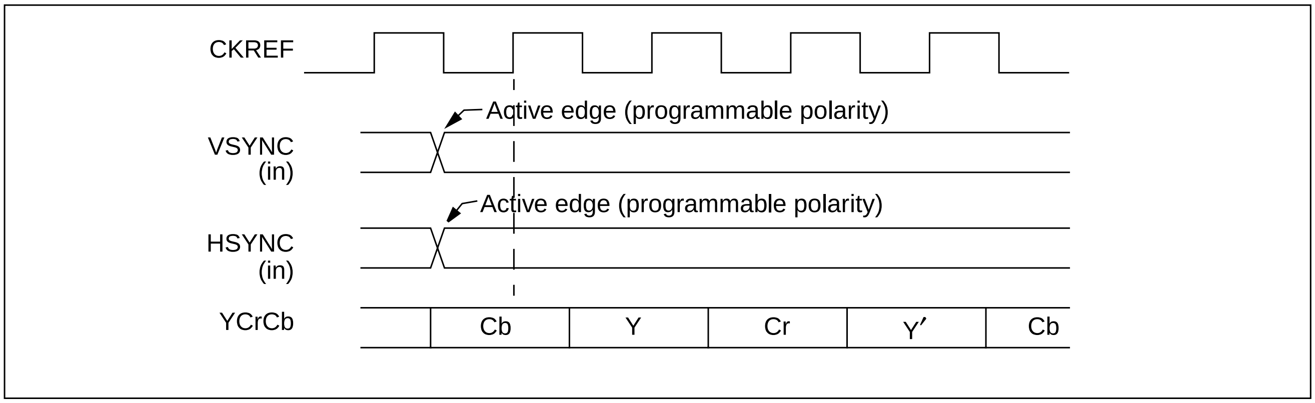

CKREF
Active edge (programmable polarity)
VSYNC (in)
Active edge (programmable polarity)
HSYNC (in)
YCrCb
Cb
Y
Cr
Y′
Cb

Figure 23.12 HSYNC + VSYNC based slave mode sync signals

Note 1: This figure is valid for bits **syncin_ad[1:0]** = default

Note 2: The active edges of HSYNC and VSYNC should normally be simultaneous. It is permissible that HSYNC transitions before VSYNC, but VSYNC must not transition before HSYNC.

### 23.5.2 Synchronization onto a frame signal

### ODDEVEN-only based synchronization

Synchronization is performed on a frame-by-frame basis by locking onto an incoming ODDEVEN signal. A line sync signal is derived internally and is also issued to the outside as HSYNC. Refer to Figure 23.13 for waveforms and timings. The phase relationship between ODDEVEN and the incoming YCrCB data is normally such that the first clock rising edge following the ODDEVEN active edge samples “Cb” (i.e. a ‘blue’ chroma sample within the YCrCb stream). It is however possible to internally delay the incoming ODDEVEN signal by up to 3 clock cycles to cope with different data/sync phasings, using configuration bits **syncin_ad** in **configuration4**.

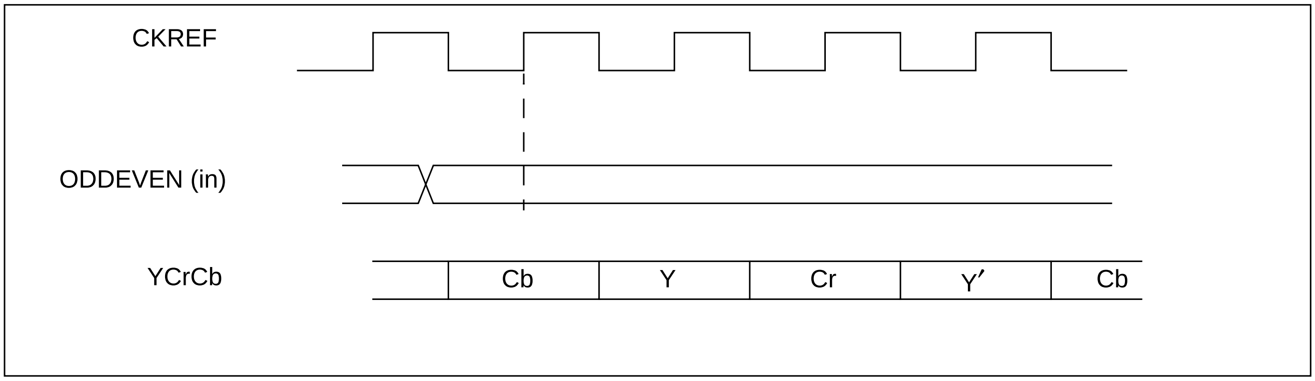


Figure 23.13 ODDEVEN based slave mode sync signals

Note: This figure is valid for bits **syncin_ad[1:0]** = default

The first active edge of ODDEVEN triggers generation of the analog sync signals and encoding of the incoming video data. Frames being supposed to be of constant duration, the next ODDEVEN active transition is expected at a precise time after the last ODDEVEN detected.

So, once an active ODDEVEN edge has been detected, checks that the following ODDEVEN are present at the expected instants are performed.

Encoding and analog sync generation carry on unless these checks fail three successive times. In that case, three behaviors are possible, according to the configuration programmed in registers **configuration1-2**:

- if **freerun** is enabled, the DENC carries on outputting the digital line sync HSYNC and generating analog video just as though the expected ODDEVEN edge had been present. However, it will re-synchronize onto the next ODDEVEN active edge detected, whatever its location.
- if **freerun** is disabled but bit **syncok** is set in the configuration registers, the DENC sets the active portion of the TV line to black level but carries on outputting the analog sync tips (on Ys and CVBS) and the digital line sync signal HSYNC. When programmed, Macrovision$^{tm}$ pseudo-sync pulses and AGC pulses are also present in the analog sync waveform.
- If **freerun** is disabled and the bit **syncok** is not set, all analog video is at black level and neither analog sync tips nor digital line sync are output.

This mode is a frame-based sync mode, as opposed to a field-based sync mode. This means that only one type of edge (rising or falling, according to programming) is of interest to the DENC; the other one is ignored.

### VSYNC-only based synchronization

Synchronization is performed on a frame-by-frame basis by locking onto an incoming VSYNC signal. An auxiliary line sync signal HSYNC must also be fed to the DENC, which uses it to reconstruct from VSYNC and HSYNC information an internal odd/even waveform identical to that of an ODDEVEN signal. Therefore the behaviour of the core is identical to that described above for ODDEVEN-only based synchronization (except that nothing is output on HSYNC pin since it is an input port in that mode).

Note that HSYNC is an input but has no other use than allowing the DENC to decide whether an incoming VSYNC pulse flags an odd or an even field. In other words, the DENC does not lock onto HSYNC in this mode since this is NOT a line-locked mode.

The phase relationship between VSYNC and the incoming YCrCb data is normally such that the first clock rising edge following the VSYNC active edge samples Cb (i.e. a blue chroma sample within the YCrCb stream). It is however possible to internally delay the incoming sync signals (VSYNCand HSYNC) by up to 3 clock cycles to cope with different data/sync phasings, using the bits **syncin_ad** in **configuration4**).

## 23.5.3 Synchronization onto data-embedded sync words

### ‘End-of-frame’ word based synchronization

Synchronization is performed by extracting the 1-to-0 transitions of the **F** flag (end-of-frame) from the **EAV** (End-of-Active-Video) sequence embedded within ITU-R656 / D1 compliant digital video streams. Both a frame sync signal and a line sync signal are derived and are made available externally as ODDEVEN and HSYNC. Refer to Figure 23.14 for waveforms and timings.

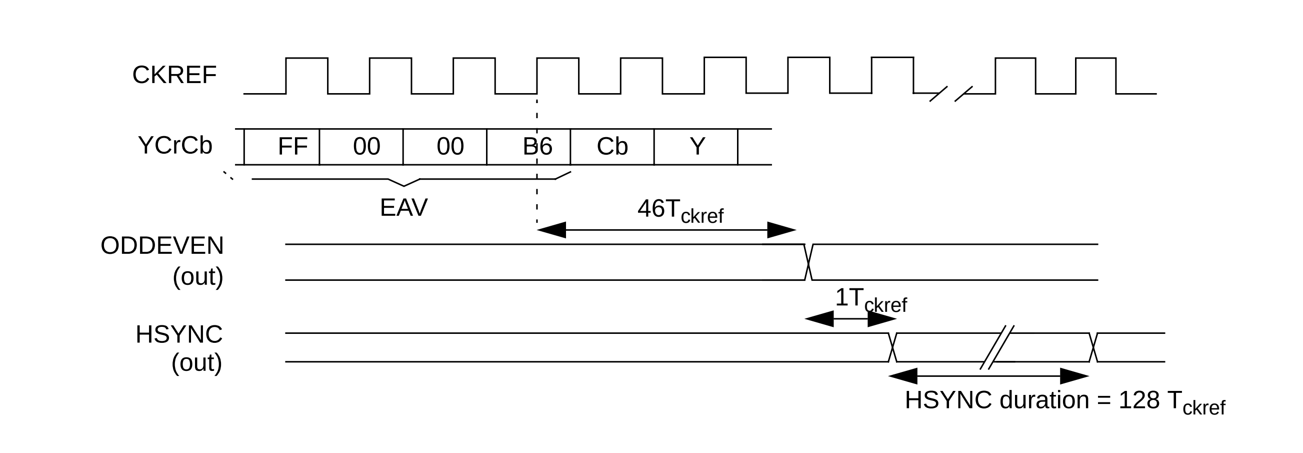


Figure 23.14 Data (EAV) based slave mode sync signals

The first successful detection of the **F** flag triggers generation of the analog sync signals and encoding of the incoming video data. Frames being supposed to be of constant duration, the next EAV word containing the **F** flag is expected at a precise time after the latest detection.

So, once an active **F** flag has been detected, checks that the following flags are present within the incoming video stream at the expected times are performed.

Encoding and analog sync generation carry on unless three successive fails of these checks occur.

In that case, three behaviours are possible, according to the configuration programmed:

- if **free-run** is enabled, the DENC carries on generating the digital frame and line SYNCs (ODDEVEN and HSYNC ) and generating analog video just as though the expected **F** flag had been present. However, it will re-synshronize onto the next **F** flag detected within the incoming ITU-R656/D1 video stream.
- if **free-run** is disabled but the bit **syncok** is set in the configuration registers, the DENC sets the active portion of the TV line to black level but carries on outputting the analog sync tips (on Ys and CVBS) and the digital frame and line sync signals ODDEVEN and HSYNC. (When programmed, Macrovision$^{TM}$ pseudo-sync pulses and AGC pulses are also present in the analog sync waveform).
- if **free-run** is disabled and the bit **syncok** is not set, all analog video is at black level and neither analog sync tips nor digital frame/line sync are output.

The SAV and EAV words are Hamming-decoded. After detection of two successive errors, a bit is set in the status register to inform the micro-controller of the poor transmission quality.

### ‘End-of-line’ word based synchronization

Synchronization is performed by extracting the **F** and **H** flags from the **SAV** (Start of Active Video) and **EAV** (End of Active Video) words embedded within ITU-R656 / D1 compliant digital video streams.

A line sync signal and a frame sync signal are derived internally from these flags and are issued to the outside on the HSYNC and ODDEVEN/VSYNC pins in output mode. These signals are also exploited by the core of the circuit which treats them like it treats incoming ODDEVEN and HSYNC signals in HSYNC+ODDEVEN based synchronization.

## 23.6 Autotest mode

An autotest mode is available, which causes the DENC to produce a color bar pattern, in the appropriate standard, independently from the video input.

The autotest mode is started by setting to 7 the 3-bit field **sync** in the register **configuration0**. As this mode sets the DENC in master mode, VSYNC/ODDEVEN and HSYNC signals are in output mode. In Table 23.2, the decimal values of Y, Cr and Cb are shown corresponding to the autotest color bar.

| | **Y** | **Cr** | **Cb** |
|---|---|---|---|
| Black | 16 | 128 | 128 |
| Blue | 36 | 116 | 212 |
| Red | 64 | 212 | 100 |
| Magenta | 84 | 200 | 184 |
| Green | 112 | 64 | 72 |
| Cyan | 136 | 44 | 156 |
| Yellow | 160 | 140 | 44 |
| White | 236 | 128 | 128 |

Table 23.2 Autotest colors

The corresponding decimal output values just before the DACs are shown graphically in Figure 23.15 and Figure 23.16. Both figures show the static values corresponding to the input values in Table 23.2.

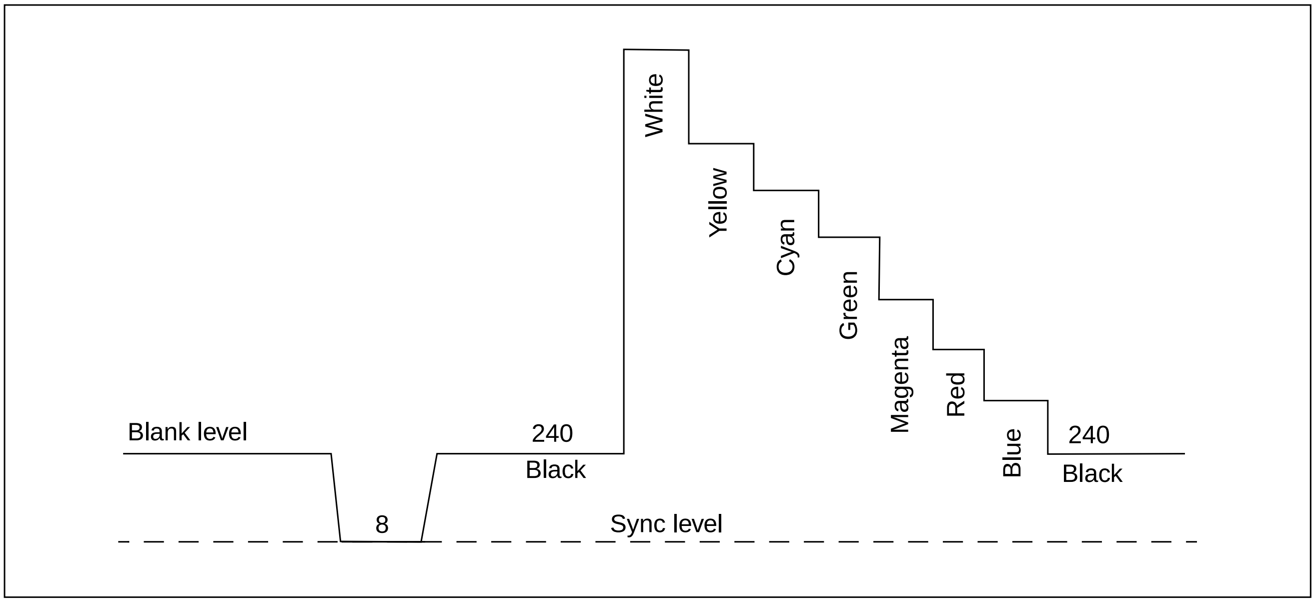


Figure 23.15 Luminance output levels in autotest for NTSC without set-up

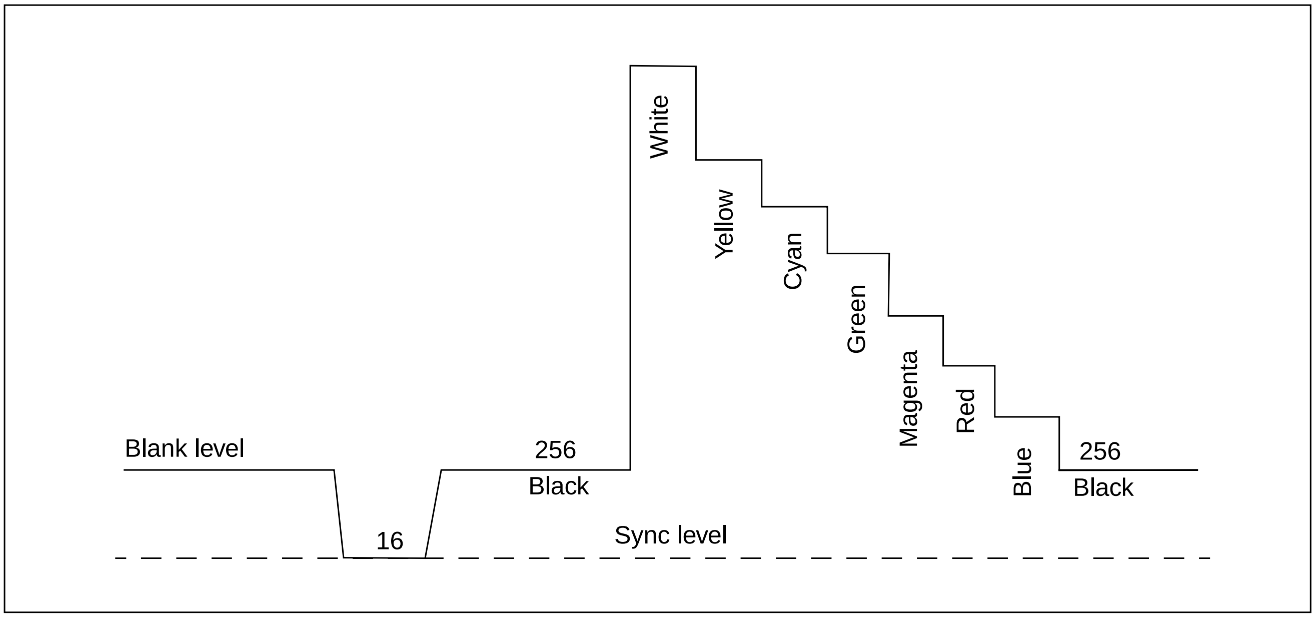


Figure 23.16 Luminance output levels in autotest for PAL (BGHI)

## 23.7 Input demultiplexor

The incoming YCrCb data is demultiplexed into a ‘blue-difference’ chroma information stream, a ‘red-difference’ chroma information stream and a luma information stream. Incoming data bits are treated as blue, red or luma samples according to their relative position with respect to the sync signals in use and the contents of configuration bits **syncin_ad** in slave modes, or **syncout_ad** in master mode.

The ITU-R601 recommendation defines the black luma level as $Y=16_{10}$ and the maximum white luma level as $Y = 235_{10}$. Similarly it defines 225 quantification levels for the color difference components (Cr, Cb), centered around 128. Accordingly, incoming YCrCB samples can be saturated in the input multiplexer with the following rules:

- for Cr or Cb samples:

| | |
|---|---|
| Cr, Cb > 240 | means that Cr, Cb are saturated at 240. |
| Cr, Cb < 16 | means that Cr, Cb are saturated at 16. |

- for Y samples:

| | |
|---|---|
| Y > 235 | means that Y is saturated at 235. |
| Y < 16 | means that Y is saturated at 16. |

This avoids having to heavily saturate the composite video codes before digital-to-analog conversion in case erroneous or unrealistic YCrCb samples are input to the encoder (there may otherwise be overflow errors in the codes driving the DACs), and therefore avoids generating a distorted output waveform.

However, in some applications, it may be desirable to let ‘extreme’ YCrCb codes pass through the demultiplexor. This is controlled using bit **maxdyn** in register **configuration6**. In this case, only codes 0x00 and 0xFF are overridden; if such codes are found in the active video samples, they are forced to 0x01 and 0xFE.

In any case, the YCrCb codes are not overridden for EAV/SAV decoder.

## 23.8 Sub-carrier generation

A Direct Digital Frequency Synthesizer (DDFS) generates the required color sub-carrier frequency using a 24-bit phase accumulator. This oscillator feeds a quadrature modulator which modulates the base-band chrominance components.

The sub-carrier frequency is obtained from the following equation:

$$\text{Fsc} = (\textit{Increment\_Word} \,/\, 2^{24}) \text{ x CKREF}$$

where *Increment _Word* is a 24-bit value. Hard-wired *Increment_Word* values are available for each standard and can be automatically selected. Alternatively in PAL and NTSC (according to bit **selrst** in **configuration2**), the frequency can be fully customized by programming other values into a dedicated *Increment_Word* register, **increment_dfs**. This allows, for instance, the encoding of NTSC-4.43 or PAL-M-4.43.

This is done with the following procedure:

- Program the required increment in **increment_dfs**.
- Set bit **selrst** to 1 in register **configuration2**.
- Perform a software reset using register **configuration6**. This sets all bits in all DENC registers except **configuration***n* to their default value.

***Warning:*** if a standard change occurs after the software reset, the increment value is automatically re-initialized with the hard wired or loaded value according to bit selrst

The reset phase of the color sub-carrier can also be software-controlled by register **phase_dfs**.

The sub-carrier phase can be periodically reset to its nominal value to compensate for any drift introduced by the finite accuracy of the calculations. In PAL and NTSC sub-carrier phase adjustment can be performed every line, every eight field, every four field, or every two field (**configuration2** bits **valrst[1:0]**).

## 23.9 Burst insertion

The color reference burst is inserted so as to always start with a positive zero crossing of the sub-carrier sine wave, except in some cases where Macrovision™ anti-copy process is active. The first and last half-cycles have a reduced amplitude so that the burst envelope starts and ends smoothly.

The burst contains 9 or 10 sine cycles of 4.43361875MHZ or 3.579545MHz (depending on the standard programmed in the register **configuration0**) as follows:

| | | |
|---|---|---|
| • NTSC-M | 9 cycles of | 3.579545MHz |
| • PAL-BDGHI | 10 cycles of | 4.43361875MHz |
| • PAL-M | 9 cycles of | 3.579545MHz |
| • PAL-N | 9 cycles of | 3.579545MHz |

It is possible to turn the burst off (no burst insertion) by setting configuration bit **bursten** to 0 in **configuration2**.

Burst insertion is performed by always starting the burst with a positive-going zero crossing. This guarantees a smooth start and end of burst with a maximum of undistorted burst cycles and can only be beneficial to chroma decoders, it is the solution implemented in the DENC.

This avoids an uncontrolled initial burst phase, and guarantees a start on a positive-going zero crossing with the consequence that two burst start locations are visible over successive lines, according to the line parity. This is normal and explained below.

In NTSC, the relation between subcarrier frequency and line length creates a $180^o$ subcarrier phase difference (with respect to the horizontal sync) from one line to the next according to the line parity. So if the burst always starts with the same phase (positive-going zero crossing), this means the burst will be inserted at time X or at time $X+T_{NTSC}/2$ after the horizontal sync tip according to the line parity, where $T_{NTSC}$ is the duration of one cycle of the NTSC burst.

With PAL, a similar rationale holds, and again there will be two possible burst start locations. The subcarrier phase difference (with respect to the horizontal sync) from one line to the next in that case is either 0 or $180^o$ with the following series: A-A-B-B-A-A-...-etc. where A denotes ‘A-type’ bursts and B denotes ‘B-type’ bursts, A-type and B-type being $180^o$ out of phase with respect to the horizontal sync. So 2 locations are possible, one for A-type, the other for B-type (see Fig 7).

This assumes a periodic reset of the subcarrier is automatically performed (see bits **valrst[1:0]** in **configuration2**). Otherwise, over several frames, the start of burst will drift within an interval of half a subcarrier’s cycle. This is normal, and means the burst is correctly locked to the colors encoded. The equivalent effect with a gated burst approach would be the following: the start location would be fixed but the phase with which the burst starts (*with respect to the horizontal sync*) would be drifting.

## 23.10 Luminance encoding

The demultiplexed Y samples are band-limited and interpolated at CKREF clock rate. The resulting luminance signal is properly scaled before insertion of any Closed-captions, CGMS or Teletext data, Macrovision copy-protection signals and synchronization pulses.

The interpolation filter compensates for the sin(x)/ x attenuation inherent in D/A conversion and greatly simplifies the output stage filter. See Figure 23.17 to Figure 23.19 for characteristic curves.

In addition, the luminance that is added to the chrominance to create the composite CVBS signal can be trap-filtered at 3.58 MHz (NTSC) or 4.43 MHz (PAL). This supports applications oriented towards low-end TV sets which are subject to cross-color if the digital source has a wide luminance bandwidth (e.g. some DVD sources). Note that the trap filter does not affect the S-VHS luminance output nor the RGB outputs.

A 7.5 IRE pedestal can be programmed if needed with all standards (see registers **configuration1** and **configuration7**). This allows in particular to encode Argentinian and non-Argentinian PAL-N, or Japanese NTSC (NTSC with no set-up).

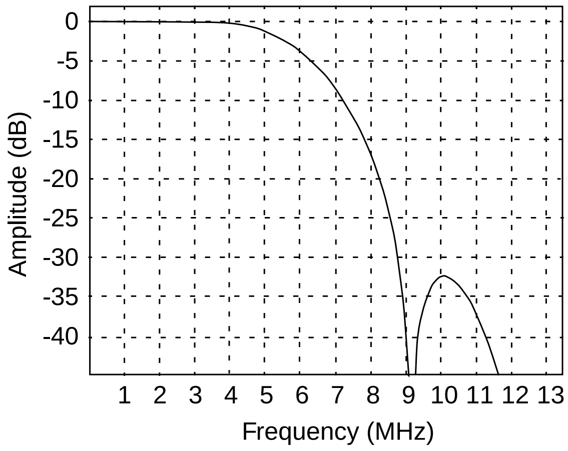


Figure 23.17 Luma filtering including DAC attenuation

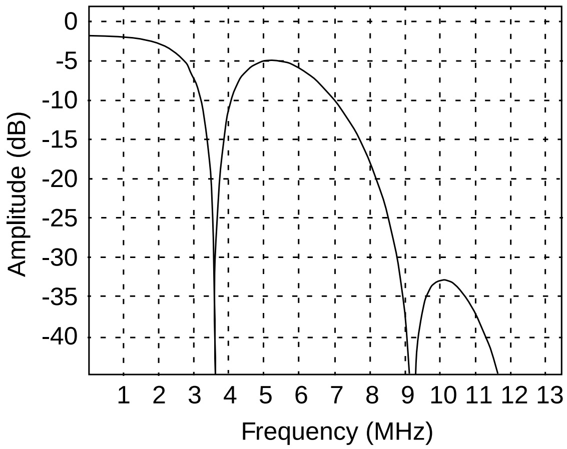


Figure 23.18 Luma filtering with 3.58MHz trap, including DAC attenuation

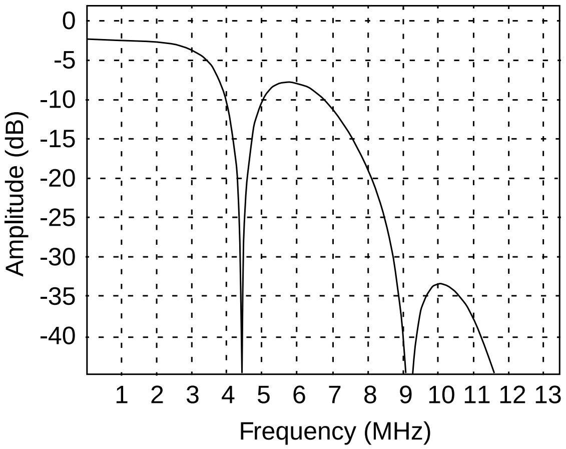


Figure 23.19 Luma filtering with 4.43MHz trap, including DAC attenuation

A programmable delay can be inserted on the luminance path to offset any chroma/luma delay introduced by off-chip filtering (chroma and luma transitions being coincident at the DAC output with default delay) (see register **configuration3**).

## 23.11 Chrominance encoding

U and V chroma components are computed from demultiplexed Cb and Cr samples. Before modulating the subcarrier, these are band-limited and interpolated at CKREF clock rate. This processing eases the filtering following D/A conversion and allows a more accurate encoding. A set of 4 different filters is available in PAL and NTSC for chroma filtering to fit a wide variety of applications in the different standards and include filters recommended by ITU-R 624-4 and SMPTE170-M. The available 3-dB bandwidths are 1.1, 1.3, 1.6 or 1.9 MHz. See Figures 23.20, 23.21, 23.22, 23.23 and 23.24 for the various frequency responses and register **configuration1** for programming.

The narrower bandwidths are useful against cross-luminance artifacts, the wider bandwidths allow higher chroma contents.

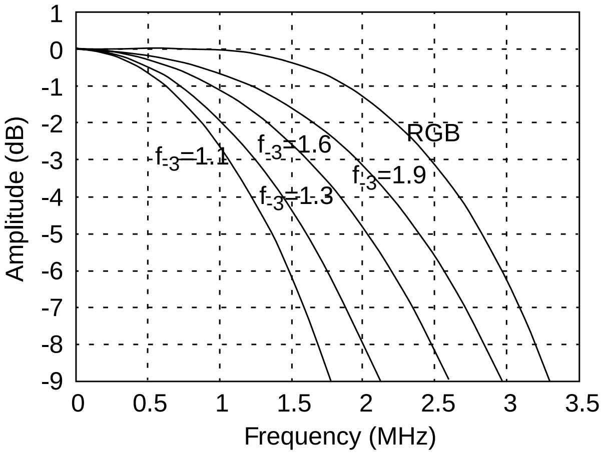


Figure 23.20 Various chroma filters available and RGB filter

## 23.12 Composite video signal generation

The composite video signal is created by adding the luminance (after trap filtering - optional in PAL and NTSC, see register **configuration3**) and the chrominance components. A saturation function is included in the adder to avoid overflow errors should extreme luminance levels be modulated with highly saturated colors. This does not occur with natural colors but may be generated by computers or graphics engines.

A ‘color killing’ function is available, whereby the composite signal contains no chrominance, i.e. replicates the trap-filtered luminance. This function does not suppress the chrominance on the S/VHS outputs, but suppressing the S-VHS chrominance is possible using bit **bdkac***n* in **configuration5**, where the chrominance signal is outputted on DAC *n*.

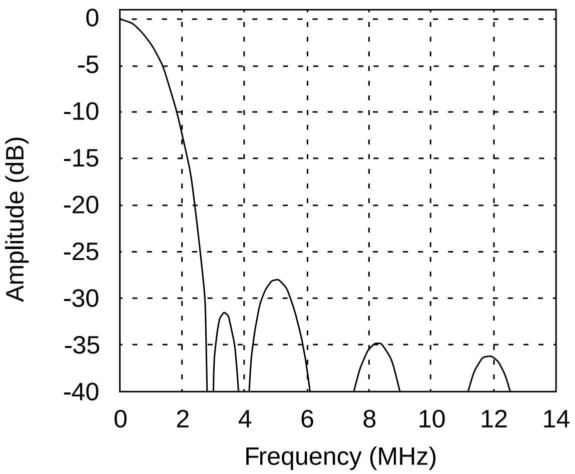


Figure 23.21 1.1 MHz chroma filter

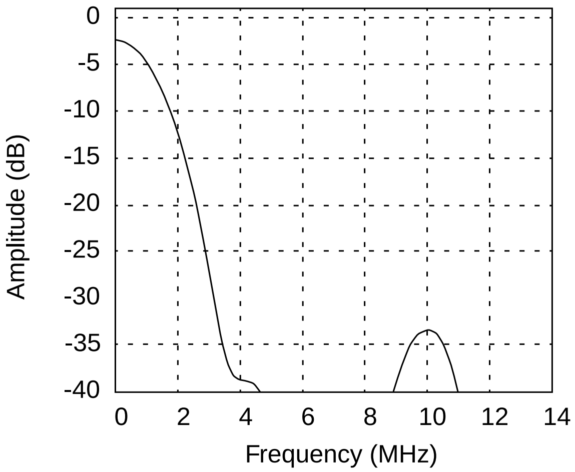


Figure 23.22 1.3 MHz chroma filter

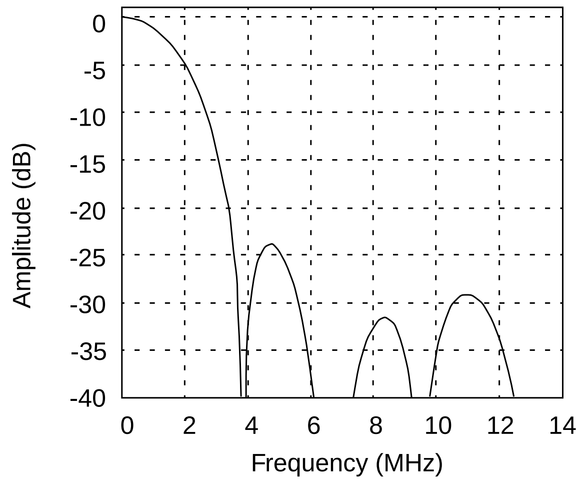


Figure 23.23 1.6 MHz chroma filter

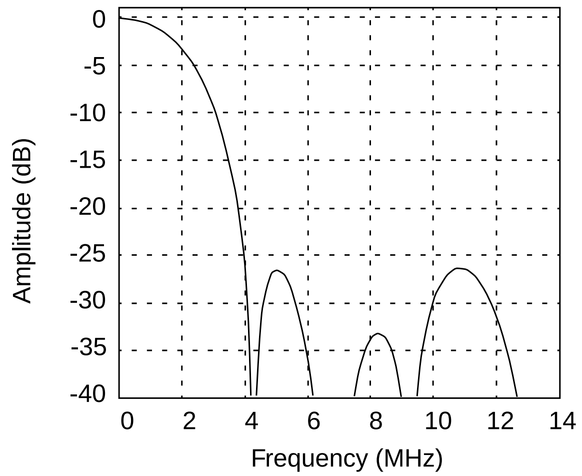


Figure 23.24  1.9 MHz chroma filter

## 23.13 RGB encoding

After demultiplexing, the Cr and Cb samples feed a 4 times interpolation filter. The resulting baseband chroma signal has a 2.45 MHz bandwidth (Figure 23.25) and is combined with the filtered luma component to generate R,G,B or U,V samples at 27 MHz.

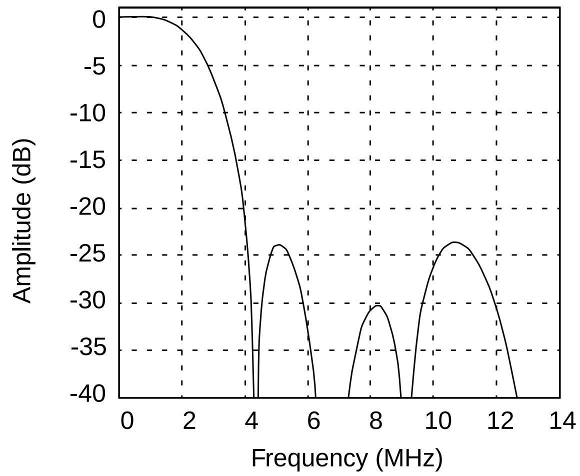


Figure 23.25 RGB - chroma filtering

## 23.14 Closed captioning

Closed-captions (or data from an Extended Data Service as defined by the Closed-Captions specification) can be encoded by the circuit. The closed caption data is delivered to the circuit through the register interface. Two dedicated pairs of bytes (two bytes per field), each pair preceded by a clock run-in and a start bit can be encoded and inserted on the luminance path on a selected TV line. The Clock Run-In and Start code are generated by the DENC.

Closed-caption data registers are double-buffered so that loading can be performed anytime, even during line 21/284 or any other selected line.

User register **cccf1** and **cccf2** each contain the first and second byte to send (LSB first) after the start bit on the appropriate TV line, where **cccf1** refers to field 1 and **cccf2** to field 2. The TV line number where data is to be encoded is programmable using registers **cclif1** and **cclif2**. Lines that may be selected include those used by the StarSight data broadcast system. Closed-captions data has priority over any CGMS or Macrovision anti-copy signals programmed for the same line.

The internal Clock Run-In generator is based on a Direct Digital Frequency Synthesizer. The nominal instantaneous data rate is 503.496 KHz (i.e. 32 times the NTSC line rate). Data LOW corresponds nominally to 0 IRE, data HIGH corresponds to 50 IRE at the DAC outputs.

When closed-captioning is on (bits **cc1** and **cc2** in **configuration1**), the CPU should load the relevant registers (**cccf1** or **cccf2**) once every frame at most (although there is in fact some margin due to the double-buffering). Two bits are set in the **status** register in case of attempts to load the closed-caption data registers too frequently; these can be used to regulate loading rate.

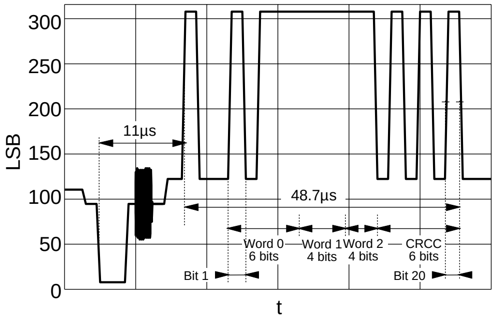


Figure 23.26 Example of closed-caption waveform

The closed captions encoder considers that closed caption data has been loaded and is valid on completion of the write operation into **cccf1** for field1, or **cccf2** for field 2. If closed caption encoding has been enabled and no new data bytes have been written into the closed caption data registers when the closed caption window starts on the appropriate TV line, then the circuit outputs two US-ASCII NULL characters with odd parity after the start bit.

## 23.15 CGMS encoding

CGMS stands for *Copy Generation Management System*, and is also known as VBID and described by standard *CPX-1204* of *EIAJ*. CGMS data can be encoded by the digital encoder.

Three bytes, containing 20 significant bits, are delivered to the chip via the register interface. Two reference bits (1 then 0) are encoded first, followed by 20 bits of CGMS data. This includes a Cyclic Redundancy Check sequence, which is not computed by the device but is supplied to it as part of the 20 data bits. The reference bits are generated locally by the DENC. Figure 23.27 shows a typical CGMS waveform.

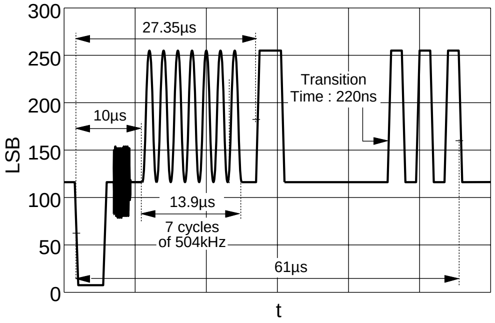


Figure 23.27 Example of CGMS waveform

CGMS encoding is enabled by setting bit **encgms** in register **configuration3**. When enabled, the CGMS waveform is present once in each field, on lines 20 and 283 (SMPTE-525 line numbering).

The CGMS data register is double-buffered, which means that it can be loaded at any time (even during line 20/283) without any risk of corrupting CGMS data that could be in the process of being encoded.The CGMS encoder considers that new CGMS data has been loaded and is valid on completion of the write operation into register **cgms**.

## 23.16 Teletext encoding

The DENC is able to encode Teletext according to the *CCIR/ITU-R Broadcast Teletext System B* specification, also known as *World System Teletext*.

In DVB applications, Teletext data is embedded within DVB streams as MPEG data packets. It is the responsibility of the software to handle incoming data packets and in particular to store Teletext packets in a buffer, which then passes them to the DENC on request.

### 23.16.1Signals exchanged

The DENC and the Teletext buffer exchange 2 signals: TTXS (Teletext Synchronization) going from the DENC to the Teletext Buffer and TTXD (Teletext Data) going from the Teletext Buffer to the DENC.

The TTXS signal is a request signal generated on selected lines. In response to this signal, the Teletext buffer is expected to send 360 Teletext bits to the DENC for insertion of a Teletext line into the analog video signal.

The duration of the TTXS window is 1402 reference clock periods (51.926 us), which corresponds to the duration of 360 Teletext bits (see Transmission Protocol below).

Following the TTXS rising edge the encoder expects data from the Teletext buffer after a programmable number (2 to 9) of 27MHz master clock periods. Data is transmitted synchronously with the master clock at an average rate of 6.9375 Mbit/s according to the protocol described below. It consists, in order of transmission, of 16 Clock Run-In bits, 8 Framing Code bits and the 336 bits (42 bytes) that represent one Teletext packet.

### 23.16.2Transmission protocol

In order to transmit the Teletext data bits at an average rate of 6.9375 Mbit/s, which is about 1 / 3.89 times the master clock frequency, the following scheme is adopted:

The 360-bit packet is regarded as nine 37-bit sequences plus one 27-bit sequence. In every sequence, each Teletext data bit is transmitted as a succession of *four* identical samples at 27 Msample/s, except for the *10th*, *19th*, *28th* and *37th* bits of the sequence which are transmitted as a succession of *three* identical samples.

### 23.16.3Programming

**‘TTXS rising’ to ‘first valid sample’ delay programming**

The encoder expects the Teletext buffer to clock out the first Teletext data sample on the $(2+N)^{th}$ rising edge of the master clock following the rising edge of TTXS. Figure 23.28 depicts this graphically for N=0.

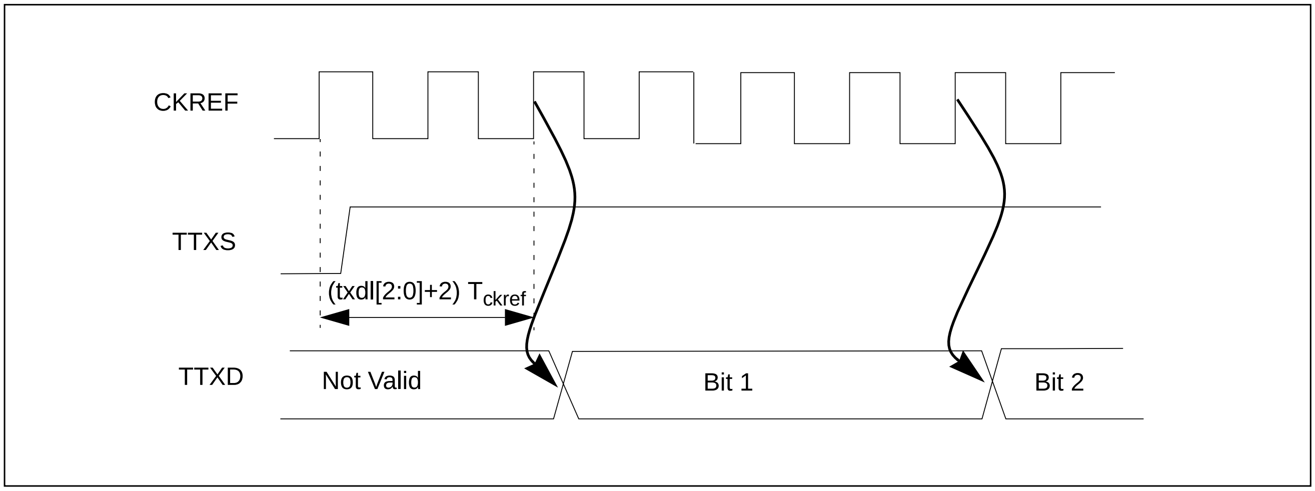


Figure 23.28 *TTXT Rising* to *First Valid Sample* delay for **txdl[2:0]** = 0

N is programmable from 0 to 7, and is written to 3 dedicated bits **txdl[2:0]** located in the register **configuration4**. The value written in **txdl[2:0]** is 2 less than the overall delay in 27MHz cycles, so a value of 0 for **txdl[2:0]** corresponds to an overall delay of 2 cycles, and a value of 7 corresponds to a delay of 9 cycles.

### Teletext line selection

Five dedicated registers allow to program Teletext encoding in various areas of the Vertical Blanking Interval (VBI) of each field. A total of 4 such areas (i.e. blocks of contiguous Teletext lines) can *independently* be defined within the two VBIs of one frame (e.g. 2 blocks in each VBI, or 3 blocks in field1 VBI and one in field2 VBI, etc.). Further, under certain circumstances, it is possible to define up to 4 areas in each VBI.

Programming is performed using four teletext block definition registers **ttx_block1-4** and a teletext block mapping register (**ttx_block_map**). Refer to the description of user registers 34 to 38 for details.

### 23.16.4Teletext pulse shape

The shape and amplitude of a single Teletext pulse are depicted in Figure 23.29, its relative power spectral density is given in Figure 23.30 and Figure 23.31 and is substantially zero at frequencies above 5 MHz, as required by the World System Teletext specification.

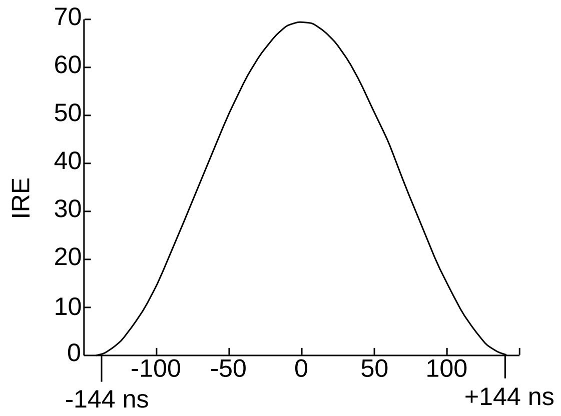


Figure 23.29 Shape and amplitude of a single teletext symbol

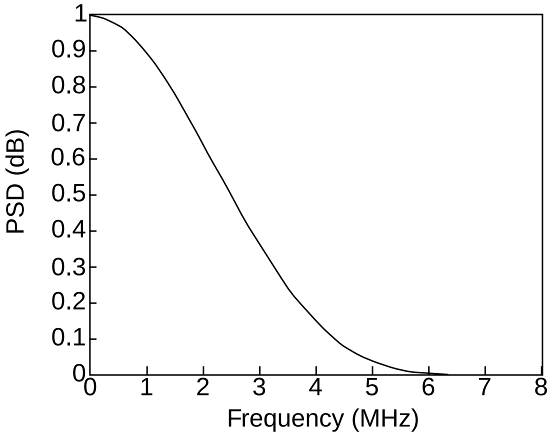
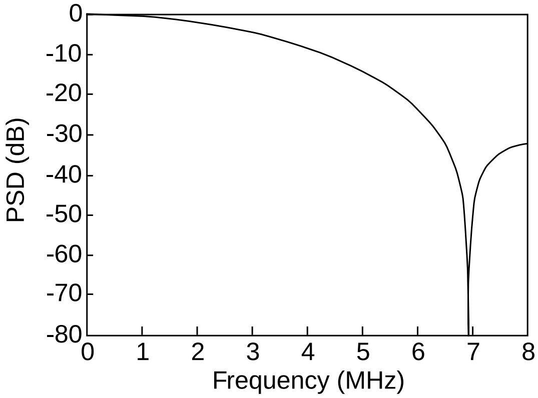


Figure 23.30 Linear PSD scale



Normalized power spectral density (PSD) of a single Teletext pulse

Figure 23.31 Logarithmic PSD scale

## 23.17 Line skip and line insert capability

This patented feature of the DENC offers the possibility to cut the cost of the application by suppressing the need for a VCXO.

Ideally, the master clock used on the application board and fed to the MPEG decoding IC would have exactly same frequency as the clock that was used when the MPEG data was encoded. Obviously this is not realistic; up to now a solution commonly used was to dynamically adjust the clock on the board as close to the 'ideal' clock as possible with the help of time stamps embedded within the MPEG stream. Such a kind of tracking often involves the use of a VCXO: when the MPEG data buffer fills up to more than some threshold the clock frequency is increased, when it empties down to some other threshold the clock frequency is lowered.

The DENC offers an alternative, cost-saving solution: by programming two bits in register **configuration6**, the DENC is able to reduce or increase the length of some frames in a way that will not introduce visible artifacts (even if comb-filtering is used). These bits should be set according to the level of the MPEG data buffer.

Operation with the DENC as sync master is as follows:

- If the MPEG data buffers fills up too much: set bit **jump** to 1 and bit **dec_ninc** to 1.The DENC will reduce the length of the current frame (Bit **jump** will then automatically reset to 0).
- If the MPEG data buffers empties too much: set bit **jump** to 1 and bit **dec_ninc** to 0. The DENC will increase the length of the current frame (Bit **jump** will then automatically reset to 0).

These operations can be repeated until the MPEG data buffer is inside its fixed limits

It is also possible to use the line skip/repeat capability in non-interlaced mode

This functionality of the DENC is also available in slave mode, in this case the sync signals supplied to the DENC must be in accordance with the modified frame lengths programmed.

## 23.18 Macrovision$^{tm}$ Copy Protection Process rev 7.01

The chrominance, luminance and composite video signals and R,G,B video signals can be altered according to the MACROVISION$^{tm}$ Copy Protection Process, Revision 7.01. This process is controlled via the parallel bus.

A programming document is available to those customers who have executed a license or a non-disclosure agreement with Macrovision Corporation.

## 23.19 CVBS, S-VHS and RGB analog outputs

Four of the six video signals (composite CVBS, S-VHS (Y/C) and RGB) can be directed to four analog output pins through 9-bit D/A converters operating at the reference clock frequency. The available combinations are:

S-VHS (Y/C) + CVBS + CVBS1, or

R, G, B + CVBS1 .

The combination is controlled by bit **rgb_nyc** in register **configuration5**.

A single external analog power supply pair is used for all DACs, but two independent pairs of current and voltage references are needed. Each current reference pin normally connects externally to a resistor tied to the analog ground, while each voltage reference pin normally connects to a capacitance tied to the analog ground.

The internal current sources are independent from the positive supply, and the consumption of the DACs is constant whatever the codes converted.

Any unused DAC may be independently disabled by software, in which case its output is at 'neutral' level (blanking for luma and composite outputs, no color for chroma output, black for RGB and UV outputs). For applications where a single CVBS output is required, the RGB/CVBS+S-VHS/UV Triple DAC should be disabled and pins **I_Ref_Dac_RGB**, **V_Ref_Dac_RGB** tied to analog power supply.

Due to the 3.3 V power supply used, the output swing of the DACs is about 1Vp-p. Therefore some external gain may be required, which, combined with the recommended output filtering stage, means active filtering. For this active filtering stage to be very simple, it is possible to 'invert' the DAC outputs by programming a bit of **configuration1**. Code 'N' becomes code '1024-N', i.e. the resulting waveform undergoes a reflection around the mid-swing code.

## 23.20 Registers

This section lists all the control registers for the digital encoder. The registers are in a block in the peripheral space in the address map. The addresses of the registers are given in this chapter as offsets from the base of this block. The base of the block is named *DENCBaseAddress*, and its value is given in the *STi5500 Register Manual*.

Table 23.3 lists the registers, giving their access type and address offset.

| Register | Type | Bits | Address | Description |
|---|---|---|---|---|
| **configuration0** | Read/write | | 0x00 | General configuration. |
| **configuration1** | Read/write | | 0x01 | General configuration. |
| **configuration2** | Read/write | | 0x02 | General configuration. |
| **configuration3** | Read/write | | 0x03 | General configuration. |
| **configuration4** | Read/write | | 0x04 | General configuration. |
| **configuration5** | Read/write | | 0x05 | General configuration. |
| **configuration6** | Read/write | | 0x06 | General configuration. |
| **status** | Read | | 0x09 | Status. |
| **increment_dfs** | Read/write | 23-16 | 0x0A | Increment for digital frequency synthesizer. |
| | | 15-8 | 0x0B | |
| | | 7-0 | 0x0C | |
| **phase_dfs** | Read/write | 23-22 | 0x0D | Static phase offset for digital frequency synthesizer. |
| | Read/write | 21-14 | 0x0E | |

Table 23.3 Register map

| Register | Type | Bits | Address | Description |
|---|---|---|---|---|
| **chipid** | Read | | 0x11 | Digital encoder identification number. |
| **revid** | Read | | 0x12 | Digital encoder revision identification number. |
| **line_reg** | Read/write | | 0x15 - 0x17 | Line jump |
| **cgms** | Read/write | 1-4 | 0x1F | CGMS data register. |
| | | 5-12 | 0x20 | |
| | | 13-20 | 0x21 | |
| **ttx_block1** | Read/write | | 0x22 | Teletext block definition (See section 23.16) |
| **ttx_block2** | Read/write | | 0x23 | |
| **ttx_block3** | Read/write | | 0x24 | |
| **ttx_block4** | Read/write | | 0x25 | |
| **ttx_block_map** | Read/write | | 0x26 | Teletext block mapping. |
| **cccf1** | Read/write | | 0x27 - 0x28 | Closed caption characters/extended data for field 1. |
| **cccf2** | Read/write | | 0x29 - 0x2A | Closed caption characters/extended data for field 2. |
| **cclif1** | Read/write | | 0x2B | Closed caption/extended data line insertion for field 1 |
| **cclif2** | Read/write | | 0x2C | Closed caption/extended data line insertion for field 2 |

Table 23.3 Register map

Table 23.4 lists the bits in each register. Bits which are not named are reserved and should be written as zero. The registers are individually described in the *STi5500 Register Manual*.

| Register | Bit 7 | Bit 6 | Bit 5 | Bit 4 | Bit 3 | Bit 2 | Bit 1 | Bit 0 |
|---|---|---|---|---|---|---|---|---|
| **configuration0** | std1 | std0 | sync2 | sync1 | sync0 | polh | polv | freerun |
| **configuration1** | blkli | flt1 | flt0 | sync_ok | coki | setup | cc2 | cc1 |
| **configuration2** | nintrl | enrst | bursten | - | selrst | rstosc | valrst1 | valrst0 |
| **configuration3** | entrap | trap_pal | encgms | nosd | del2 | del1 | del0 | - |
| **configuration4** | syncin_ ad1 | syncin_ad0 | syncout_ ad1 | syncout_ ad0 | aline | ttxdel2 | ttxdel1 | ttxdel0 |
| **configuration5** | - | bkcvbs | bkys | bkc | bkr | bkg | bkb | dacinv |
| **configuration6** | softreset | jump | dec_ninc | free_jump | cfc1 | cfc0 | - | maxdyn |
| **status** | hok | atfr | b2_free | b1_free | fldct2 | fldct1 | fldct0 | jump |
| **increment_dfs** | d23 | d22 | d21 | d20 | d19 | d18 | d17 | d16 |
| | d15 | d14 | d13 | d12 | d11 | d10 | d9 | d8 |
| | d7 | d6 | d5 | d4 | d3 | d2 | d1 | d0 |
| **phase_dfs1** | - | - | - | - | - | - | o23 | o22 |
| **phase_dfs2** | o21 | o20 | o19 | o18 | o17 | o16 | o15 | o14 |
| **line_reg** | ltarg8 | ltarg7 | ltarg6 | ltarg5 | ltarg4 | ltarg3 | ltarg2 | ltarg1 |

Table 23.4 Register bits

| Register | Bit 7 | Bit 6 | Bit 5 | Bit 4 | Bit 3 | Bit 2 | Bit 1 | Bit 0 |
|---|---|---|---|---|---|---|---|---|
| **line_reg** | ltarg0 | lref8 | lref7 | lref6 | lref5 | lref4 | lref3 | lref2 |
| **line_reg** | lref1 | lref0 | - | - | - | - | - | - |
| **chipid** | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 |
| **revid** | Revision | | | | | | | |
| **cgms1** | - | - | - | - | bit1 | bit2 | bit3 | bit4 |
| **cgms2** | bit5 | bit6 | bit7 | bit8 | bit9 | bit10 | bit11 | bit12 |
| **cgms3** | bit13 | bit14 | bit15 | bit16 | bit17 | bit18 | bit19 | bit20 |
| **ttx_block1** | ttxbs1.3 | ttxbs1.2 | ttxbs1.1 | ttxbs1.0 | ttxbe1.3 | ttxbe1.2 | ttxbe1.1 | ttxbe1.0 |
| **ttx_block2** | ttxbs2.3 | ttxbs2.2 | ttxbs2.1 | ttxbs2.0 | ttxbe2.3 | ttxbe2.2 | ttxbe2.1 | ttxbe2.0 |
| **ttx_block3** | ttxbs3.3 | ttxbs3.2 | ttxbs3.1 | ttxbs3.0 | ttxbe3.3 | ttxbe3.2 | ttxbe3.1 | ttxbe3.0 |
| **ttx_block4** | ttxbs4.3 | ttxbs4.2 | ttxbs4.1 | ttxbs4.0 | ttxbe4.3 | ttxbe4.2 | ttxbe4.1 | ttxbe4.0 |
| **ttx_block_map** | ttxbmf1.1 | ttxbmf1.2 | ttxbmf1.3 | ttxbmf1.4 | ttxbmf2.1 | ttxbmf2.2 | ttxbmf2.3 | ttxbmf2.4 |
| **cccf1** | opc11 | c117 | c116 | c115 | c114 | c113 | c112 | c111 |
| **cccf1** | opc12 | c127 | c126 | c125 | c124 | c123 | c122 | c121 |
| **cccf2** | opc21 | c217 | c216 | c215 | c214 | c213 | c212 | c211 |
| **cccf2** | opc22 | c227 | c226 | c225 | c224 | c223 | c222 | c221 |
| **cclif1** | - | - | - | l1_4 | l1_3 | l1_2 | l1_1 | l1_0 |
| **cclif2** | - | - | - | l2_4 | l2_3 | l2_2 | l2_1 | l2_0 |

Table 23.4  Register bits

# 24 SDRAM block move

This module copies blocks of data from one byte address within the SDRAM to another. A general purpose block move engine, for transfering to and from any address, is described in Chapter 26.

A source address, a destination address and a count of the number of bytes to be transferred must be specified.

## 24.1 Moving blocks of data

To perform a SDRAM block move, from one SDRAM memory buffer to another, the SDRAM block move module must first be initialised by writing to three registers in a fixed order. The write to the third register initiates the block move, which completes without further CPU intervention.

The source, destination and size of the blocks must be 64-bit aligned, i.e. the byte addresses and the size in bytes must be divisible by 8.

While a block move is in progress, all other accesses to SDRAM are disabled. The progress of the block move can be monitored using **VID_STA.BMI**; bit **VID_STA.BMI** is set while a block move is in progress, and reset when the block move engine is idle. When the block move completes, an interrupt will be generated if the mask bit **VID_ITM.BMI** is set.

Three registers are provided, a block move size register, a read pointer and a write pointer, as listed in Table 24.1. The registers are all serial read/write registers in the peripheral address space.

| Register | Bits | Cycles | Name |
|---|---|---|---|
| **USD_BMS** | 15:0 | 2 | Block move size register. |
| **USD_BWP** | 20:0 | 3 | Memory write pointer. |
| **USD_BRP** | 20:0 | 3 | Memory read pointer. |

Table 24.1 SDRAM block move registers

# Part E Peripherals

# 25 Clocks

This chapter describes the various clocks provided on the STi5500 for clocking subsystems and for system and programmer use. Clock recovery is described in Chapter 16.

The STi5500 has two on-chip phase locked loops (PLLs) to generate all the required high frequency internal clocks for device operation from a single 27 MHz input clock, **PixClk_27Mhz**:

- ST20 clock PLL generating the processor and peripheral 50 MHz clocks.
- MPEG/system clock PLL. This clock is programmable and can be used to generate many different clocks by fractional division. It also provides the audio decoder system clock and the transport stream demultiplexor and front end clock.

The single clock input **PixClk_27Mhz** must be 27 MHz for PLL operation.

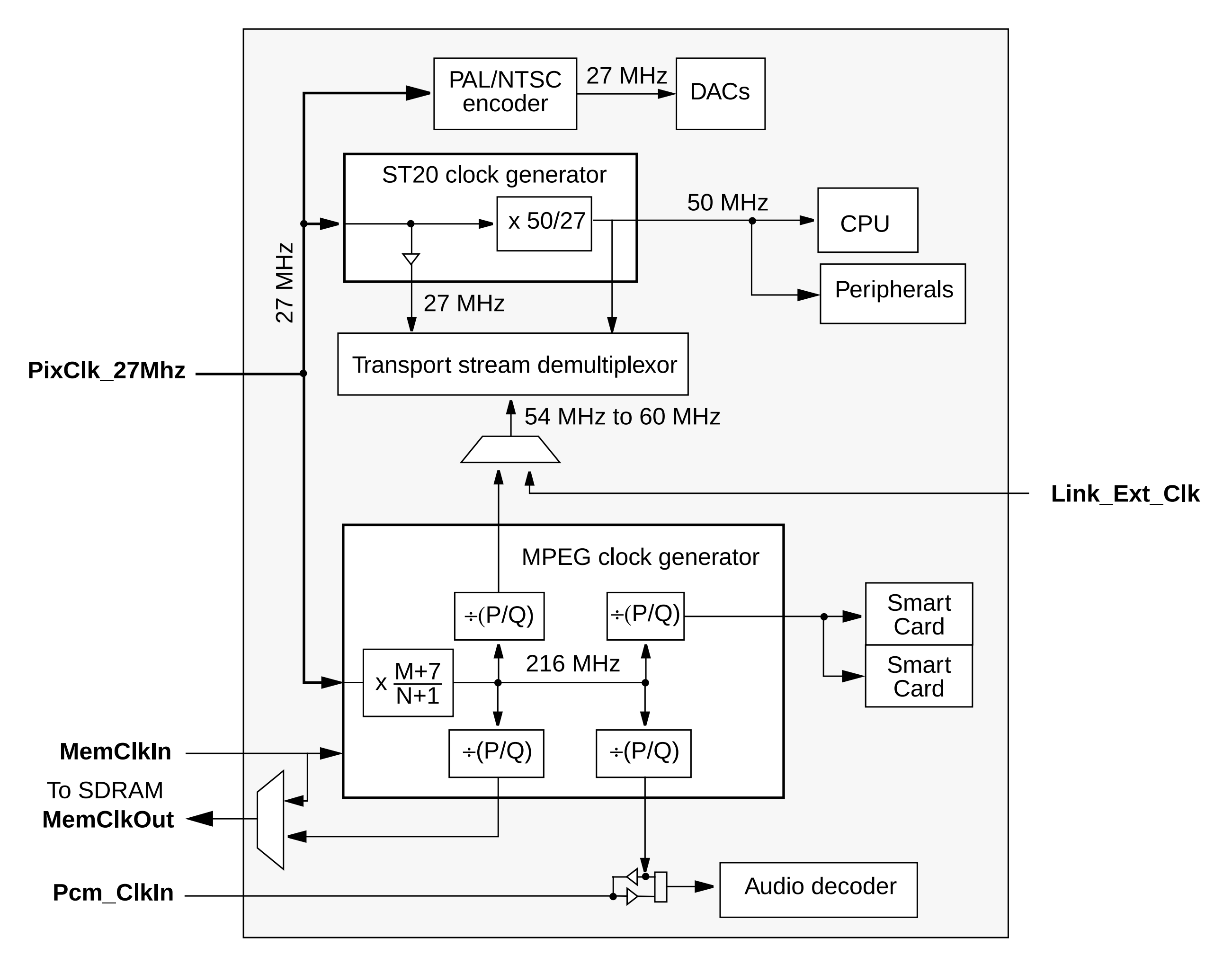


Figure 25.1 STi5500 clocks generation

Figure 25.1 shows the configuration of all the PLLs, with nominal clock speeds, and the clock distribution within the device.

## 25.1 ST20 clock

The ST20 clock PLL is used to generate the internal clock frequencies needed for the CPU and the OS-Link. The PLL is hard wired to produce the clock frequencies shown in Table 25.2 when the frequency of **PixClk_27Mhz** is 27 MHz.

| Speed Select1:0 | Processor clock speed (MHz) | Processor cycle time ns | High priority timer (MHz) | Low priority timer (MHz) | Link speed Mbits/s |
|---|---|---|---|---|---|
| 00 | Times one mode | | | | |
| 01 | 60.0 | 16.67 | 1.0 | 0.015625 | 19.20 |
| 10 | 39.9 | 25.06 | 0.9975 | 0.015586 | 19.95 |
| 11 | 49.875 | 20.05 | 0.9975 | 0.015586 | 19.95 |

Table 25.1 Processor speed selection with 27MHz **PixClk_27MHz**

| Nominal speed | Processor clock speed | Processor cycle time | Phase lock loop factor (PLLx) | OS-Link speed | Approx. high priority timer period | Approx. low priority timer period |
|---|---|---|---|---|---|---|
| 50 MHz | 49.875 MHz | 20.05 ns | 1.85 | 19.95 Mbit/s | 1.0025 µs | 64.16 µs |

Table 25.2 Processor speed selection

## 25.2 MPEG/system clock

The MPEG/system clock generation consists of a patented frequency synthesizer circuit and fractional dividers which derive all of the required system clocks from a single selectable input, thus eliminating the need for external dividers and PLL circuitry.

The reference input frequency can be the incoming **PixClk_27Mhz** or the PCM clock. The selected reference clock frequency is multiplied by a programmable integrated PLL and the output of the PLL is steered to a bank of programmable fractional dividers to generate the following output clocks:

- the internal MPEG audio decoder and PCM clock;
- the SDRAM memory and video decoder clock;
- the transport stream demultiplexor clock;
- the SmartCard clock, which goes to both SmartCards.

An external PCM clock can be provided as an alternative to the internal audio PCM clock.

The internal video decoder clocks are generated from the SDRAM memory clock by division.

Each of the dividers is controlled by a dedicated register and there are two general configuration registers. These registers are described in detail in the register manual, and are summarized in Table 25.3.

| Register | Type | Address | Function |
|---|---|---|---|
| **CKG_PLL** | R/W | 0x30 | PLL parameters. Controls the selection of the PLL multiplying factor and reference frequency. |
| **CKG_CFG** | R/W | 0x31 | Clock configuration. Controls the selection of either fixed ratio, fractional divider or external clock where appropriate. |
| **CKG_LNK** | R/W | 0x33 | Transport stream demultiplexor clock. Fractional divider programming. |
| **CKG_MCK** | R/W | 0x36 | SDRAM clock. Fractional divider programming. |
| **CKG_SMC** | R/W | 0x32 | Smart Card clock. Fractional divider programming. |
| **CKG_PCM** | R/W | 0x35 | PCM and audio decoder clock. Fractional divider programming. |
| **CKG_PXC** | R/W | 0x34 | Pixel clock. Fractional divider programming. |

Table 25.3 MPEG / system clock registers

### 25.2.1 Programming the fractional dividers

The fractional dividers perform a division by P/Q. This can be expressed as:

$$\frac{P}{Q} = P_0 + 1 + \frac{P_r}{Q} \qquad \text{Equation 1}$$

where:

$$5 \le \frac{P}{Q} \le 17 \qquad \text{Equation 2}$$

$$2 \le P_0 \le 15 \qquad \text{Equation 3}$$

$$0 \le P_r \le 1023 \qquad \text{Equation 4}$$

$$0 \le Q \le 2047 \qquad \text{Equation 5}$$

$$P_r \le Q \qquad \text{Equation 6}$$

$$\frac{P_r}{Q} = 1\,,\ Q = 0 \qquad \text{Equation 7}$$

Thus, the output frequency of each fractional divider, $f_{OUT}$, is calculated as:

$$f_{OUT} = \frac{f_{VCO}}{P_0 + 1 + \frac{P_r}{Q}} \qquad \text{Equation 8}$$

or

$$f_{OUT} = \frac{f_{IN} \times \frac{M+7}{N+1}}{P_0 + 1 + \frac{P_r}{Q}} \qquad \text{Equation 9}$$

The values for M and N are programmed in register **CKG_PLL**, with M in the range 0-7 and N in the range 0 or 1. $P_0$, $P_r$ and Q are programmable for each of the fractional dividers and are stored in clock generator registers.

# 26 Block move DMA

This module copies blocks of data from one byte address anywhere in memory to another. The maximum transfer size is 65535 bytes. The transfer is performed by a dedicated DMA engine, and does not require any CPU intervention once the transfer has started. No polling or interrupt handlers are required.

The interface to the block move module is provided using the channel model, as described in Appendix A. An output instruction is executed by the CPU referring to the *Block Move DMA Controller Channel*. The output instruction initiates the transfer and deschedules the software process containing the instruction. The process is rescheduled automatically when the transfer is complete, so no interrupt is required.

A source address, a destination address and a count of the number of bytes to be transferred must be specified. The base address for the output buffer in the memory space, from which the block move source data is taken, and the size of the transfer in bytes are set by the *out* instruction. For the mapping of channels into the address space, see Chapter 8.

## 26.1 Moving blocks of data

To perform a DMA block move from one memory buffer to another, the block move module must first be initialized and then an output to the block move channel executed by the CPU.

The register **BMDmaAddress** holds the destination address. This register must be written with the address of the first byte of the destination buffer before each transfer. This must be done before every transfer because after the transfer the value is left undefined.

The final stage of initializing the block move DMA transfer is to execute an output to the block move DMA channel. The output is performed by the ST20 *out* instruction. The parameters of this instruction are:

- the address of the *Block Move DMA Controller Channel*, loaded into the **Creg**;
- the base address of the data to be moved, loaded into the **Breg**;
- the DMA transfer size in bytes, loaded into the **Areg**.

The *Block Move DMA Controller Channel* is at the fixed address #80000034 in the system space at the bottom of the memory map. Executing the *out* instruction with this channel initiates the block move and de-schedules the software process until the transfer is complete.

The block move DMA controller fetches 64-bit blocks as pairs of words from the source block whenever possible. It buffers the bytes before performing word writes in pairs to the destination block. The blocks to be moved do not need to be word aligned.

When the number of bytes programmed in the *out* instruction have been transferred, the channel output is acknowledged to the CPU and the software process is rescheduled. This means that when the next instruction after the *out* is executed, the block move is complete.

# 27 PWM and counter module

This module provides three PWM encoder outputs, three PWM decoder (capture) inputs and four programmable timers. Each capture input can be programmed to detect rising edge, falling edge, both edges or neither edge (disabled). These facilities are clocked by two independent clocks, one for PWM outputs and one for capture inputs/timers.

The PWM module contains four PWMs, numbered 0 to 3. In the STi5500, not all the facilities described in this chapter can be used, since some of the interface pins of the module are not available as external pins of the device. In particular:

- the timer compare output from PWM2 is not connected to a pin, but is connected internally as an internal watchdog reset.
- only the capture input of PWM3 is connected to an external pin, **CaptureIn3**. This can be used to generate an interrupt.

The module is programmed by means of registers described in the individual sections.

The module generates a single interrupt signal. The exact event which caused an interrupt can be determined by reading status bits in a register, which can then be cleared.

## 27.1 External interface

The PWM pins are listed in Table 27.1. Some of thesepins are shared with the PIO ports, as shown in Table 27.1 and described in Chapter 3.

| Pin | Shared pin | In/Out | Function |
|---|---|---|---|
| **Brm0** | | out | PWM outputs |
| **Brm1** | **BootFromROM** | | |
| **Brm2** | **Oslink_Sel** | | |
| **CaptureIn0** | **PIO2[7]** | in | Capture trigger inputs |
| **CaptureIn1** | **PIO1[3]** | | |
| **CaptureIn2** | **PIO1[4]** | | |
| **CaptureIn3** | **PIO4[6]** | | |
| **CompareOut0** | **PIO2[4]** | out | Compare output |
| **CompareOut1** | **PIO4[5]** | | |

Table 27.1 PWM and counter pins

## 27.2 PWM outputs

There are four PWM outputs which share a common counter. The relative width (in counts) of the output pulse on pin **PWMOut***N* is set between 1 and 256 by loading a value from 0 to 255 into the register **PWMVal***N.* The width cannot be less than 1, and if it is 256 the pin is continuously high. Pulses occur every 256 counts.

The counter is clocked by the 27MHz clock **ClockIn** divided by a prescaler. The prescaling factor, and therefore the period represented by one count, is determined by the value of field **PWMClkValue** in register **Control**. The factor can be from 1 to 16.

The counter (in register **PWMCount**) is enabled by setting the **PWMEnable** bit of the **Control** register to 1. When it is disabled (**PWMEnable** is 0), **PWMOut** is forced low. **PWMCount** is writable at any time but can have a synchronization latency.

When the PWM counter overflows, an interrupt is generated if the **PWMIntEn** bit of the **PWMIntEnable** register is set to 1. Bit **PWMInt** of register **PWMIntStatus** becomes 1, and can be reset by writing 1 to bit **PWMIntAck** of register **PWMIntAck**.

### 27.2.1 Registers

The PWM and counter module is programmable using control registers which are mapped into the peripheral address space. The registers for the PWM module are grouped in a 4 Kbyte block. The base of the register block is at the address *PWMBaseAddress*, where the value of *PWMBaseAddress* is given in the *Memory Map* chapter. The addresses of the registers are given in the tables as offsets from this address.

#### Pulse width

| **PWMVal*N*** | | **PWMBaseAddress + offset** (see below) **Read/Write** |
|---|---|---|
| **Bit** | **Bit field** | **Function** |
| 8:0 | **PWMVal*N*** | 8-bit pulse width (width = value + 1 counts) |

Table 27.2 **PWMVal*N*** register format

| Register | Offset from PWMBaseAddress |
|---|---|
| **PWMVal0** | #00 |
| **PWMVal1** | #04 |
| **PWMVal2** | #08 |
| **PWMVal3** | #0C |

Table 27.3 **PWMVal*N*** register offsets

#### Interrupt enable

| **PWMIntEnable** | | **PWMBaseAddress + #54** **Read/Write** |
|---|---|---|
| **Bit** | **Bit field** | **Function** |
| 0 | **PWMIntEn** | PWM counter overflow interrupt enable |

Table 27.4 **PWMIntEnable** register format (PWM outputs field)

### Interrupt status

| **PWMIntStatus** | | **PWMBaseAddress + #58** | **Read** |
|---|---|---|---|
| **Bit** | **Bit field** | **Function** | |
| 0 | **PWMInt** | PWMInt = 1 identifies the interrupt as having been caused by PWM counter overflow | |

Table 27.5 **PWMIntStatus** register format (PWM outputs field)

### Interrupt acknowledge

| **PWMIntAck** | | **PWMBaseAddress + #5C** | **Write** |
|---|---|---|---|
| **Bit** | **Bit field** | **Function** | |
| 0 | **PWMIntAck** | Interrupt acknowledge: write 1 to reset PWMInt to 0 | |

Table 27.6 **PWMIntAck** register format (PWM outputs field)

### Control register

| **PWMControl** | | **PWMBaseAddress + #50** | **Read/Write** |
|---|---|---|---|
| **Bit** | **Bit field** | **Function** | |
| 3:0 | **PWMClkValue** | PWM clock prescale factor 0-15 (divide clock by value + 1) | |
| 9 | **PWMEnable** | Enables PWM counter when = 1 | |

Table 27.7 **PWMControl** register format (PWM outputs fields)

### PWM output counter

| **PWMCount** | | **PWMBaseAddress + #60** | **Read/Write (but see text)** |
|---|---|---|---|
| **Bit** | **Bit field** | **Function** | |
| 7:0 | **PWMCount** | PWM output counter | |

Table 27.8 **PWMCount** register format

## 27.3 Capture inputs

There are four capture inputs which share a common counter with four compare facilities.

What constitutes an event on input **CaptureIn**_N_ is defined by the code in register **CaptureEdge**_N_. Possible events are rising edge, falling edge, both or neither (in other words, disabled).

When an input event occurs on input **CaptureIn**_N_, the value of the counter (in register **PWMCaptureCount**) at that time is captured in register **PWMCaptureVal**_N_. The value can be #00000000 to #FFFFFFFF.

When an input event occurs, an interrupt is generated provided the **Capture**_N_**IntEn** bit of the **PWMIntEnable** register is set to 1. Bit **CaptureInt**_N_ of register **PWMIntStatus** becomes 1, and can be reset by writing 1 to bit **CaptureIntAck**_N_ of register **PWMIntAck**.

The counter is not stopped nor reset by any of these events. See section 27.5 for details.

## Capture event definition

| **PWMCaptureEdge***N* | **PWMBaseAddress + offset** (see below) | **Read/Write** |
|---|---|---|
| **Bit** | **Bit field** | **Function** |
| 1:0 | **CaptureEdge***N* | 01 = rising edge, 10 = falling edge, 11 = rising or falling edge, 00 = disabled |

Table 27.9 **PWMCaptureEdge***N* register format

| **Register** | **Offset from PWMBaseAddress** |
|---|---|
| **PWMCaptureEdge0** | #30 |
| **PWMCaptureEdge1** | #34 |
| **PWMCaptureEdge2** | #38 |
| **PWMCaptureEdge3** | #3C |

Table 27.10 **PWMCaptureEdge***N* register offsets

## Capture value

| **PWMCaptureVal*N*** | **PWMBaseAddress + offset** (see below) | **Read only** |
|---|---|---|
| **Bit** | **Bit field** | **Function** |
| 31:0 | **CaptureVal***N* | 32-bit counter value at time of event occurring at CaptureInN |

Table 27.11 **PWMCaptureVal***N* register format

| **Register** | **Offset from PWMBaseAddress** |
|---|---|
| **PWMCaptureVal0** | #10 |
| **PWMCaptureVal1** | #14 |
| **PWMCaptureVal2** | #18 |
| **PWMCaptureVal3** | #1C |

Table 27.12 **PWMCaptureVal***N* register offsets

## Interrupt enable

| **PWMIntEnable** | **PWMBaseAddress + #54** | **Read/Write** |
|---|---|---|
| **Bit** | **Bit field** | **Function** |
| 1 | **CaptureIntEn0** | Capture 0 interrupt enable: 1 = enabled |
| 2 | **CaptureIntEn1** | Capture 1 interrupt enable: 1 = enabled |
| 3 | **CaptureIntEn2** | Capture 2 interrupt enable: 1 = enabled |
| 4 | **CaptureIntEn3** | Capture 3 interrupt enable: 1 = enabled |

Table 27.13 **PWMIntEnable** register format (Capture fields)

### Interrupt status

| **PWMIntStatus** | | **PWMBaseAddress + #58** | **Read** |
|---|---|---|---|
| **Bit** | **Bit field** | **Function** | |
| 1 | **CaptureInt0** | Capture 0 interrupt: 1 = interrupt | |
| 2 | **CaptureInt1** | Capture 1 interrupt: 1 = interrupt | |
| 3 | **CaptureInt2** | Capture 2 interrupt: 1 = interrupt | |
| 4 | **CaptureInt3** | Capture 3 interrupt: 1 = interrupt | |

Table 27.14 **PWMIntStatus** register format (Capture fields)

### Interrupt acknowledge

| **PWMIntAck** | | **PWMBaseAddress + #5C** | **Write** |
|---|---|---|---|
| **Bit** | **Bit field** | **Function** | |
| 1 | **CaptureIntAck0** | Capture 0 interrupt acknowledge: write 1 to reset status bit | |
| 2 | **CaptureIntAck1** | Capture 1 interrupt acknowledge: write 1 to reset status bit | |
| 3 | **CaptureIntAck2** | Capture 2 interrupt acknowledge: write 1 to reset status bit | |
| 4 | **CaptureIntAck3** | Capture 3 interrupt acknowledge: write 1 to reset status bit. | |

Table 27.15 **PWMIntAck** register format (Capture fields)

## 27.4 Compare (programmable timer) facilities

There are four programmable timer facilities which share a common counter with four capture inputs.Each of four compare registers **PWMCompareVal***N* in the module can be set to a value #00000000 to #FFFFFFFF.

When the counter in register **PWMCaptureCount** reaches the value of register **PWMCompareVal***N*, two things happen:

1 An interrupt is generated provided the **PWMCompareIntEn***N* bit of the **PWMIntEnable** register is set to 1. Bit **PWMCompareInt***N* of register **PWMIntStatus** becomes 1, and can be reset by writing 1 to bit **PWMCompareIntAck***N* of register **PWMIntAck**.

2 Pin **PWMCompareOut***N* takes on the value set in register **PWMCompareOutVal***N.*

The counter is not stopped nor reset by any of these events. See section 27.5 below for details of the counter.

### Compare value

| **PWMCompareVal***N* | | **PWMBaseAddress + offset** (see below) | **Read/Write** |
|---|---|---|---|
| **Bit** | **Bit field** | **Function** | |
| 31:0 | **CompareVal***N* | 32-bit counter value at time of event occurring at **CompareIn***N* | |

Table 27.16 **PWMCompareVal***N* register format

| Register | Offset from PWMBaseAddress |
|---|---|
| **PWMCompareVal0** | #20 |
| **PWMCompareVal1** | #24 |
| **PWMCompareVal2** | #28 |
| **PWMCompareVal3** | #2C |

Table 27.17 **PWMCompareVal***N* register offsets

### Interrupt enable

| **PWMIntEnable** | **PWMBaseAddress + #54** | **Read/Write** |
|---|---|---|
| **Bit** | **Bit field** | **Function** |
| 5 | **CompareIntEn0** | Compare 0 interrupt enable: 1 = enabled |
| 6 | **CompareIntEn1** | Compare 1 interrupt enable: 1 = enabled |
| 7 | **CompareIntEn2** | Compare 2 interrupt enable: 1 = enabled |
| 8 | **CompareIntEn3** | Compare 3 interrupt enable: 1 = enabled |

Table 27.18 **PWMIntEnable** register format (Compare fields)

### Interrupt status

| **PWMIntStatus** | **PWMBaseAddress + #58** | **Read** |
|---|---|---|
| **Bit** | **Bit field** | **Function** |
| 5 | **CompareInt0** | Compare 0 interrupt: 1 = interrupt |
| 6 | **CompareInt1** | Compare 1 interrupt: 1 = interrupt |
| 7 | **CompareInt2** | Compare 2 interrupt: 1 = interrupt |
| 8 | **CompareInt3** | Compare 3 interrupt: 1 = interrupt |

Table 27.19 **PWMIntStatus** register format (Compare fields)

### Interrupt acknowledge

| **PWMIntAck** | **PWMBaseAddress + #5C** | **Write** |
|---|---|---|
| **Bit** | **Bit field** | **Function** |
| 5 | **CompareIntAck0** | Compare 0 interrupt acknowledge: write 1 to reset status bit |
| 6 | **CompareIntAck1** | Compare 1 interrupt acknowledge: write 1 to reset status bit |
| 7 | **CompareIntAck2** | Compare 2 interrupt acknowledge: write 1 to reset status bit |
| 8 | **CompareIntAck3** | Compare 3 interrupt acknowledge: write 1 to reset status bit |

Table 27.20 **PWMIntAck** register format (Compare fields)

### Compare output value

| **PWMCompareOutVal*N*** | **PWMBaseAddress + offset** (see below) | **Read/Write** |
|---|---|---|
| **Bit** | **Bit field** | **Function** |
| 0 | **CompareOutVal*N*** | Value written to **CompareOut*N*** pin when compare value = counter value |

Table 27.21 **PWMCompareOutVal***N* register format

| **Register** | **Offset from PWMBaseAddress** |
|---|---|
| **PWMCompareOutVal0** | #40 |
| **PWMCompareOutVal1** | #44 |
| **PWMCompareOutVal2** | #48 |
| **PWMCompareOutVal3** | #4C |

Table 27.22 **PWMCompareOutVal*N*** register offsets

## 27.5 Capture/compare counter, prescaling and clocking

The capture/compare counter is clocked from the prescaled system clock, and is common to all capture and compare functions. The prescaling factor, and therefore the period represented by one count, is determined by the value of field **CaptureClkValue** in register **PWMControl**. The factor can be from 1 to 32.

The counter (in register **PWMCaptureCount**) is enabled by setting the **PWMCaptureEnable** bit of the **Control** register to 1. When it is disabled (**PWMCaptureEnable** is 0), none of the capture or compare functions work. **PWMCaptureCount**, like **PWMCount**, can be read or written at any time.

When the capture/compare counter reaches its maximum count of #FFFFFFFF, it wraps round to count up from zero again.

### Control: Clock Prescaling, Counter Enables

| **PWMControl** | **PWMBaseAddress + #50** | **Read/Write** |
|---|---|---|
| **Bit** | **Bit field** | **Function** |
| 8:4 | **CaptureClkValue** | Capture/compare clock prescale factor 0-31 (divide clock by value + 1) |
| 10 | **CaptureEnable** | Enables capture/compare counter when = 1 |

Table 27.23 **PWMControl** register format (capture/compare fields)

### Capture/Compare Counter

| **PWMCaptureCount** | **PWMBaseAddress + #64** | **Read/Write** |
|---|---|---|
| **Bit** | **Bit field** | **Function** |
| 31:0 | **CaptureCount** | Capture/compare counter |

Table 27.24 **PWMCaptureCount** register format

# 28 Asynchronous serial controller

The Asynchronous Serial Controller (ASC), also referred to as the UART interface, provides serial communication between the STi5500 and other microcontrollers, microprocessors or external peripherals. The STi5500 provides four ASCs.

Eight or nine bit data transfer, parity generation, and the number of stop bits are programmable. Parity, framing, and overrun error detection is provided to increase the reliability of data transfers. Transmission and reception of data is double-buffered. For multiprocessor communication, a mechanism to distinguish the address from the data bytes is included. Testing is supported by a loopback option. A 16-bit baud rate generator provides the ASC with a separate serial clock signal.

Each ASC supports full-duplex asynchronous communication, where both the transmitter and the receiver use the same data frame format and the same baud rate. Data is transmitted on the transmit data output pin **TXD** and received on the receive data input pin **RXD**.

Each ASC can be set to operate in SmartCard mode for use when interfacing to a SmartCard.

## 28.1 Control

Each ASC can be controlled by registers which are mapped into the device address space. The registers for each ASC are grouped in a 4 Kbyte block, with the base of the block for ASC number *n* at the address *ASC*n*BaseAddress*. The value of *ASC*n*BaseAddress* is given in the *Memory Map* chapter. The addresses of the registers are given in the tables as offsets from this address. Table 28.1 lists the registers for each ASC.

| Register | Address offset | Function | Bits | Access | Reset value |
|---|---|---|---|---|---|
| **ASC*n*BaudRate** | #00 | Baud rate generator/reload. | 16 | R/W | - |
| **ASC*n*TxBuffer** | #04 | Output buffer. | 9 | W | - |
| **ASC*n*RxBuffer** | #08 | Input buffer. | 9 | R | - |
| **ASC*n*Control** | #0C | Control register. | 10 | R/W | 0 |
| **ASC*n*IntEnable** | #10 | Enable interrupts. | 6 | R/W | - |
| **ASC*n*Status** | #14 | Interrupt status. | 6 | R | - |
| **ASC*n*Guardtime** | #18 | Delay before assertion of transmitter empty flag. | 8 | R/W | - |

Table 28.1 ASC registers

### 28.1.1 Control register

The **ASC*n*Control** register controls the operating mode of the ASC and contains control bits for mode and error check selection, and status flags for error identification. The format of the register is shown in Table 28.2.

Programming the mode control field (**Mode**) to one of the reserved combinations may result in unpredictable behavior. Serial data transmission or reception is only possible when the baud rate generator run bit (**Run**) is set to 1. When the **Run** bit is set to 0, **TXD** will be 1. Setting the **Run** bit to 0 will immediately freeze the state of the transmitter and receiver. This should only be done when the ASC is idle.

| **ASC*n*Control** | **ASC*n*BaseAddress + #0C** | **Read/Write** |
|---|---|---|
| **Bit** | **Bit field** | **Function** |
| 2:0 | **Mode** | ASC mode control:<br>**Mode2:0** **Mode**<br>000 RESERVED.<br>001 8-bit data.<br>010 RESERVED.<br>011 7-bit data + parity.<br>100 9-bit data.<br>101 8-bit data + wake up bit.<br>110 RESERVED.<br>111 8-bit data + parity. |
| 4:3 | **StopBits** | Number of stop bits selection:<br>**StopBits1:0** **Number of stop bits**<br>00 0.5 stop bits.<br>01 1 stop bits.<br>10 1.5 stop bits.<br>11 2 stop bits. |
| 5 | **ParityOdd** | Parity selection:<br>0 Even parity (parity bit set on odd number of ‘1’s in data).<br>1 Odd parity (parity bit set on even number of ‘1’s in data). |
| 6 | **LoopBack** | Loopback mode enable bit:<br>0 Standard transmit/receive mode.<br>1 Loopback mode enabled. |
| 7 | **Run** | Baudrate generator run bit:<br>0 Baudrate generator disabled (ASC inactive).<br>1 Baudrate generator enabled. |
| 8 | **RxEnable** | Receiver enable bit:<br>0 Receiver disabled.<br>1 Receiver enabled. |
| 9 | **SCEnable** | SmartCard enable bit:<br>0 SmartCard mode disabled.<br>1 SmartCard mode enabled. |
| 15:10 | | Reserved. Write 0. Read back 0. |

Table 28.2 **ASC*n*Control** register format

## 28.2 Transmission and reception

### 28.2.1 Buffer registers

Serial data transmission or reception is only possible when the baud rate generator run bit (**Run**) is set to 1. A transmission is started by writing to the transmit buffer register **ASC*n*TxBuffer**, defined in Table 28.3.

Data transmission is double-buffered, so a new character may be written to the transmit buffer register before the transmission of the previous character is complete. This allows characters to be sent back-to-back without gaps.

| **ASC*n*TxBuffer** | **ASC*n*BaseAddress + #04** | **Write only** |
|---|---|---|
| **Bit** | **Bit field** | **Function** |
| 0 | **TD0** | Transmit buffer data **D0.** |
| 1 | **TD1** | Transmit buffer data **D1.** |
| 2 | **TD2** | Transmit buffer data **D2.** |
| 3 | **TD3** | Transmit buffer data **D3.** |
| 4 | **TD4** | Transmit buffer data **D4.** |
| 5 | **TD5** | Transmit buffer data **D5.** |
| 6 | **TD6** | Transmit buffer data **D6.** |
| 7 | **TD7/Parity** | Transmit buffer data **D7**, or parity bit - dependent on the operating mode (the setting of the **Mode** field of the **ASCControl** register). |
| 8 | **TD8/Parity /Wake/0** | Transmit buffer data **D8**, or parity bit, or wake-up bit or undefined - dependent on the operating mode (the setting of the **Mode** field of the **ASCControl** register).<br>Note: If the Mode field selects an 8 bit frame then this bit should be written as 0. |
| 15:9 | | Reserved. Write 0. |

Table 28.3 **ASC*n*TxBuffer** register format

Data reception is enabled by the receiver enable bit (**RxEnable**) in the **ASC*n*Control** register. After reception of a character has been completed, the received data and, if provided by the selected operating mode, the received parity bit can be read from the receive buffer register (**ASC*n*RxBuffer**), defined in Table 28.4.

| **ASC*n*RxBuffer** | | **ASC*n*BaseAddress + #08** | **Read only** |
|---|---|---|---|
| **Bit** | **Bit field** | **Function** | |
| 0 | **RD0** | Receive buffer data **D0.** | |
| 1 | **RD1** | Receive buffer data **D1.** | |
| 2 | **RD2** | Receive buffer data **D2.** | |
| 3 | **RD3** | Receive buffer data **D3.** | |
| 4 | **RD4** | Receive buffer data **D4.** | |
| 5 | **RD5** | Receive buffer data **D5.** | |
| 6 | **RD6** | Receive buffer data **D6.** | |
| 7 | **RD7/Parity** | Receive buffer data **D7**, or parity bit - dependent on the operating mode (the setting of the **Mode** bit of the **ASCControl** register). | |
| 8 | **RD8/Parity/ Wake/X** | Receive buffer data **D8**, or parity bit, or wake-up bit - dependent on the operating mode (the setting of the **Mode** field of the **ASCControl** register)<br>Note: If the Mode field selects an 8 bit frame then this bit is undefined. Software should ignore this bit when reading 8 bit frames | |
| 15:9 | | Reserved. Read back 0. | |

Table 28.4 **ASC*n*RxBuffer** register format

Data reception is double-buffered, so that reception of a second character may begin before the received character has been read out of the receive buffer register. The overrun error status flag (**OverrunError**) in the status register (**ASC*n*Status**) (see Table 28.7) will be set when the receive buffer register has not been read by the time reception of a second character is complete. The previously received character in the receive buffer is overwritten, and the **ASC*n*Status** register is updated to reflect the reception of the new character.

The loop-back option (selected by the **LoopBack** bit) internally connects the output of the transmitter shift register to the input of the receiver shift register. This may be used to test serial communication routines at an early stage without having to provide an external network.

### 28.2.2 Data frames

Data frames may be 8-bit or 9-bit, with or without parity and with or without a wake-up bit. The data frame type is selected by the setting of the **Mode** bit field in the **ASC*n*Control** register, see Table 28.2.

The transmitted data frame consists of three basic elements:

- the start bit;
- the data field (8 or 9 bits, least significant bit (LSB) first, including a parity bit or wake-up bit, if selected);
- the stop bits (0.5, 1, 1.5 or 2 stop bits).

#### 8-bit data frames

Figure 28.1 illustrates an 8-bit data frame. 8-bit frames may use of one of the following formats:

- eight data bits **D0-7** (**Mode** set to 001);

- seven data bits **D0-6** plus an automatically generated parity bit (**Mode** set to 011).

Parity may be odd or even, depending on the **ParityOdd** bit in the **ASC*n*Control** register. If the modulo 2 sum of the seven data bits is 1, then the even parity bit will be set and the odd parity bit will be cleared. The parity error flag (**ParityError**) will be set if a wrong parity bit is received. The parity bit itself will be stored in bit 7 of the **ASC*n*RxBuffer** register.

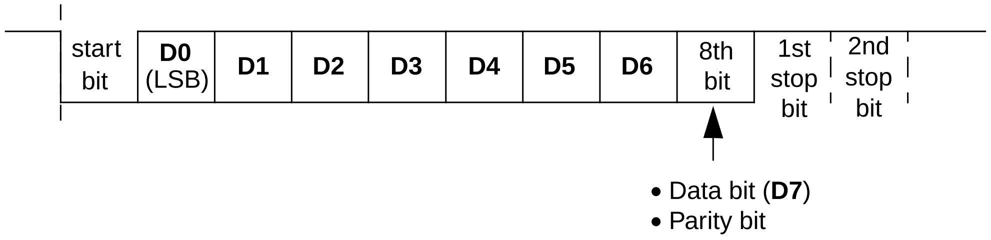


Figure 28.1 8-bit data frames

### 9-bit data frames

Figure 28.2 illustrates a 9-bit data frame. 9-bit data frames use of one of the following formats:

- nine data bits **D0-8** (**Mode** set to 100);
- eight data bits **D0-7** plus an automatically generated parity bit (**Mode** set to 111);
- eight data bits **D0-7** plus a wake-up bit (**Mode** set to 101).

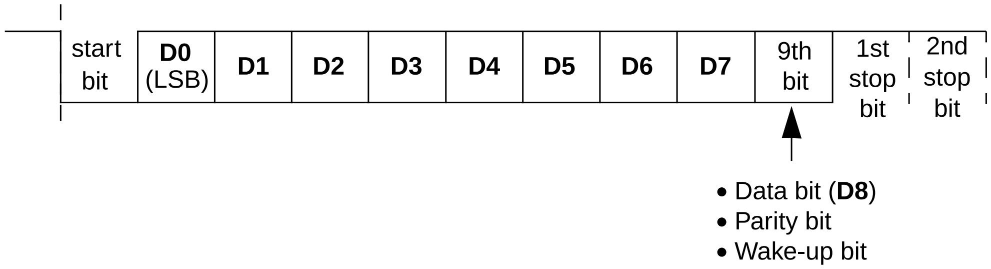


Figure 28.2 9-bit data frames

Parity may be odd or even, depending on the **ParityOdd** bit in the **ASC*n*Control** register. If the modulo 2 sum of the eight data bits is 1, then the even parity bit will be set and the odd parity bit will be cleared. The parity error flag (**ParityError**) will be set if a wrong parity bit is received. The parity bit itself will be stored in the ninth bit of the **ASC*n*RxBuffer** register, as shown in Table 28.4.

In wake-up mode, received frames are only transferred to the receive buffer register if the ninth bit (the wake-up bit) is 1. If this bit is 0, no receive interrupt request will be activated and no data will be transferred.

This feature may be used to control communication in multi-processor systems. When the master processor wants to transmit a block of data to one of several slaves, it first sends out an address byte which identifies the target slave. An address byte differs from a data byte in that the additional ninth bit is a 1 for an address byte and a 0 for a data byte, so no slave will be interrupted by a data byte. An address byte will interrupt all slaves (operating in *8-bit data plus wake-up bit* mode), so each slave can examine the 8 least significant bits (LSBs) of the received character, which is the address. The addressed slave will switch to 9-bit data mode, which enables it to receive the data bytes that will be coming (with the wake-up bit cleared). The slaves that are not being addressed remain in *8-bit data plus wake-up bit* mode, ignoring the data bytes which follow.

### 28.2.3 Transmission

Transmission begins at the next overflow of the divide-by-16 counter, provided that the **Run** bit is set and data has been loaded into the **ASC*n*TxBuffer**.

Data transmission is double buffered. When the transmitter is idle, the transmit data written into the transmit buffer **ASC*n*TxBuffer** is immediately moved to the transmit shift register, thus freeing the transmit buffer for the next data to be sent. This is signalled by the transmit buffer empty flag (**TxBufEmpty**) being set. The transmit buffer can thus be loaded with the next data, while the previous data is being transmitted.

The transmitter empty flag (**TxEmpty**) indicates whether the output shift register is empty. It will be set at the beginning of the last data frame bit that is transmitted, i.e. during the first system clock cycle of the first stop bit shifted out of the transmit shift register.

The loop-back option (selected by the **LoopBack** bit of the **ASC*n*Control** register) internally connects the output of the transmitter shift register to the input of the receiver shift register. This may be used to test serial communication routines at an early stage without having to provide an external network.

### 28.2.4 Reception

Reception is initiated by a falling edge on the data input pin **RXD**, provided that the **Run** and **RxEnable** bits of the **ASC*n*Control** register are set. The **RXD** pin is sampled at 16 times the rate of the selected baud rate. A majority decision of the first, second and third samples of the start bit determines the effective bit value. This avoids erroneous results that may be caused by noise.

If the detected value of the first bit of a frame is not a 0, then the receive circuit is reset and waits for the next falling edge transition at the **RXD** pin. If the start bit is valid, i.e. is 0, the receive circuit continues sampling and shifts the incoming data frame into the receive shift register. For subsequent data and parity bits, the majority decision of the seventh, eighth and ninth samples in each bit time is used to determine the effective bit value. The effective values received on **RxD** are shifted into a 10-bit input shift register.

For 0.5 stop bits, the majority decision of the third, fourth, and fifth samples during the stop bit is used to determine the effective stop bit value. For 1 and 2 stop bits, the majority decision of the seventh, eighth, and ninth samples during the stop bits is used to determine the effective stop bit values. For 1.5 stop bits, the majority decision of the fifteenth, sixteenth, and seventeenth samples during the stop bits is used to determine the effective stop bit value.

When the last stop bit has been received (at the end of the last programmed stop bit period) the content of the receive shift register is transferred to the receive data buffer register (**ASC*n*RxBuffer**). The receive buffer full flag (**RxBufFull**) is set, and the parity (**ParityError**) and framing error (**FrameError**) flags are updated at the same time, after the last stop bit has been received, i.e. at the end of the last stop bit programmed period. The flags are updated even if no valid stop bits have been received. The receive circuit then waits for the next falling edge transition at the **RXD** pin.

Reception is stopped by clearing the **RxEnable** bit of **ASC*n*Control**. Any currently received frame is completed including the generation of the receive status flags. Start bits that follow this frame will not be recognized.

The most significant bit of each input entry records whether or not there was a frame error when that entry was received (i.e. one of the effective stop bit values was ‘0’). The **FrameError** bit of the **ASC*n*Status** register is set when at least one of the valid entries in the input buffering has its most significant bit set.

If the mode is one where a parity bit is expected, then the next bit records whether there was a parity error when that entry was received. It does not contain the parity bit that was received. The **ParityError** bit of **ASC*n*Status** is set when at least one of the valid entries in the input FIFO has bit 8 set.

## 28.3 Hardware error detection capabilities

To improve the safety of serial data exchange, the ASC provides three error status flags in the **ASC*n*Status** register which indicate if an error has been detected during reception of the last data frame and associated stop bits.

- The parity error bit (**ParityError**) in the **ASC*n*Status** register is set when the parity check on the received data is incorrect.
- The framing error bit (**FrameError**) in the **ASC*n*Status** register is set when the **RXD** pin is not a 1 during the programmed number of stop bit times, sampled as described in the section above.
- The overrun error bit (**OverrunError**) in the **ASC*n*Status** register is set when the last character received in the **ASC*n*RxBuffer** register has not been read out before reception of a new frame is complete.

These flags are updated simultaneously with the transfer of data to the receive buffer.

## 28.4 Baud rate generation

Each ASC has its own dedicated 16-bit baud rate generator with 16-bit reload capability.

The baud rate generator is clocked with the CPU clock. The timer counts downwards and can be started or stopped by the **Run** bit in the **ASC*n*Control** register. Each underflow of the timer provides one clock pulse. The timer is reloaded with the value stored in its 16-bit reload register each time it underflows.

### 28.4.1 Baud rate generator register

The **ASC*n*BaudRate** register is the dual-function baud rate generator and reload value register. A read from this register returns the content of the timer; writing to it updates the reload register.

| **ASC*n*BaudRate** | | **ASC*n*BaseAddress + #00** | **Read/Write** |
|---|---|---|---|
| **Bit** | **Bit field** | **Write Function** | **Read Function** |
| 15:0 | **ReloadVal** | 16-bit reload value. | 16-bit count value. |

Table 28.5 **ASC*n*BaudRate** register format

If the **Run** bit of the control register is 1, then any value written in the **ASC*n*BaudRate** register is immediately copied to the timer. However, if the **Run** bit is 0 when the register is written, then the timer will not be reloaded until the first CPU clock cycle after the **Run** bit is 1.

### 28.4.2 Baud rates

The baud rate generator provides a clock at 16 times the baud rate. This clock only ticks if the **Run** bit of the **ASC*n*Control** register is set to 1. Setting this bit to 0 will immediately freeze the state of the ASCs transmitter and receiver.

The baud rate and the required reload value for a given baud rate can be determined by the following formulae:

$$\mathit{BaudRate} = \frac{f_{CPU}}{16 \text{ x } \mathit{ASCBaudRate}}$$

$$\mathit{ASCBaudRate} = \frac{f_{CPU}}{16 \text{ x } \mathit{BaudRate}}$$

where: *ASCBaudRate* represents the content of the **ASC*n*BaudRate** register, taken as an unsigned 16-bit integer,
$f_{CPU}$ is the frequency of the CPU.

Table 28.6 lists commonly used baud rates with the required reload values and the approximate deviation errors for an example baud rate with a CPU clock of 50 MHz. This does not imply availability of a 50 MHz device.

| Baud rate | Reload value (exact) | Reload value (integer) | Reload value (hex) | Approx. deviation error |
|---|---|---|---|---|
| 625 K | 5 | 5 | 0005 | 0% |
| 38.4 K | 81.380 | 81 | 0051 | 0.1% |
| 19.2 K | 162.760 | 163 | 00A3 | 0.1% |
| 9600 | 325.521 | 325 | 0145 | 0.2% |
| 4800 | 651.042 | 651 | 028B | 0.01% |
| 2400 | 1302.083 | 1302 | 0516 | 0.01% |
| 1200 | 2604.167 | 2604 | 0A2C | 0.01% |
| 600 | 5208.33 | 5208 | 1458 | 0.01% |
| 300 | 10416.667 | 10417 | 28B1 | 0.01% |
| 75 | 41666.667 | 41667 | A2C3 | 0.01% |

Table 28.6 Baud rates

## 28.5 Interrupt control

Each ASC contains two registers that are used to control interrupts, the status register (**ASC*n*Status**) and the interrupt enable register (**ASC*n*IntEnable**). The status bits in the **ASC*n*Status** register show the cause of any interrupt. The interrupt enable register allows certain interrupt causes to be masked. Interrupts will occur when a status bit is 1 (high) and the corresponding bit in the **ASC*n*IntEnable** register is 1.

The error interrupt signal is generated by the ASC from the OR of the parity error, framing error, and overrun error status bits after they have been ANDed with the corresponding enable bits in the **ASC*n*IntEnable** register. An overall interrupt request signal is generated from the OR of the error interrupt signal and the **TxEmpty**, **TxBufEmpty** and **RxBufFull** signals, as shown in Figure 28.3.

Software cannot write directly to the status register. The reset mechanism for the status register is described below. The transmitter interrupt status bits (**TxEmpty**, **TxBufEmpty**) are reset when a character is written to the transmitter buffer. The receiver interrupt status bit (**RxBufFull**) is reset when a character is read from the receive buffer. The error status bits (**ParityError**, **FrameError**, **OverrunError**) are reset when a character is read from the receive buffer.

| **ASC*n*Status** | **ASC*n*BaseAddress + #14** | **Read Only** |
|---|---|---|
| **Bit** | **Bit field** | **Function** |
| 0 | **RxBufFull** | Receiver buffer full flag:<br>0 Receiver buffer is not full.<br>1 Receiver buffer is full. |
| 1 | **TxEmpty** | Transmitter empty flag:<br>0 Ttransmitter is not empty.<br>1 Transmitter is empty. |
| 2 | **TxBufEmpty** | Transmitter buffer empty flag:<br>0 Transmitter buffer not empty.<br>1 Transmitter buffer empty. |
| 3 | **ParityError** | Input parity error flag:<br>0 No parity error.<br>1 Parity error. |
| 4 | **FrameError** | Input frame error flag, i.e.stop bits not found:<br>0 No framing error.<br>1 Framing error. |
| 5 | **OverrunError** | Overrun error flag:<br>0 No overrun error.<br>1 Overrun error, i.e. data received when the input buffer is full. |
| 7:6 |  | Reserved. Read 0. |

Table 28.7 **ASC*n*Status** register format

| **ASC*n*IntEnable** | | **ASC*n*BaseAddress + #10** **Read/Write** |
|---|---|---|
| **Bit** | **Bit field** | **Function** |
| 0 | **RxBufFullIE** | Receiver buffer full interrupt enable:<br>0 Receiver buffer full interrupt disable.<br>1 Receiver buffer full interrupt enable. |
| 1 | **TxEmptyIE** | Transmitter empty interrupt enable:<br>0 Transmitter empty interrupt disable.<br>1 Transmitter empty interrupt enable. |
| 2 | **TxBufEmptyIE** | Transmitter buffer empty interrupt enable:+<br>0 Transmitter buffer empty interrupt disable.<br>1 Transmitter buffer empty interrupt enable. |
| 3 | **ParityErrorIE** | Parity error interrupt enable:<br>0 Parity error interrupt disable.<br>1 Parity error interrupt enable. |
| 4 | **FrameErrorIE** | Framing error interrupt enable:<br>0 Framing error interrupt disable.<br>1 Framing error interrupt enable. |
| 5 | **OverrunErrorIE** | Overrun error interrupt enable:<br>0 Overrun error interrupt disable.<br>1 Overrun error interrupt enable. |
| 7:6 | | Reserved. Write 0. Read back 0. |

Table 28.8 **ASC*n*IntEnable** register format

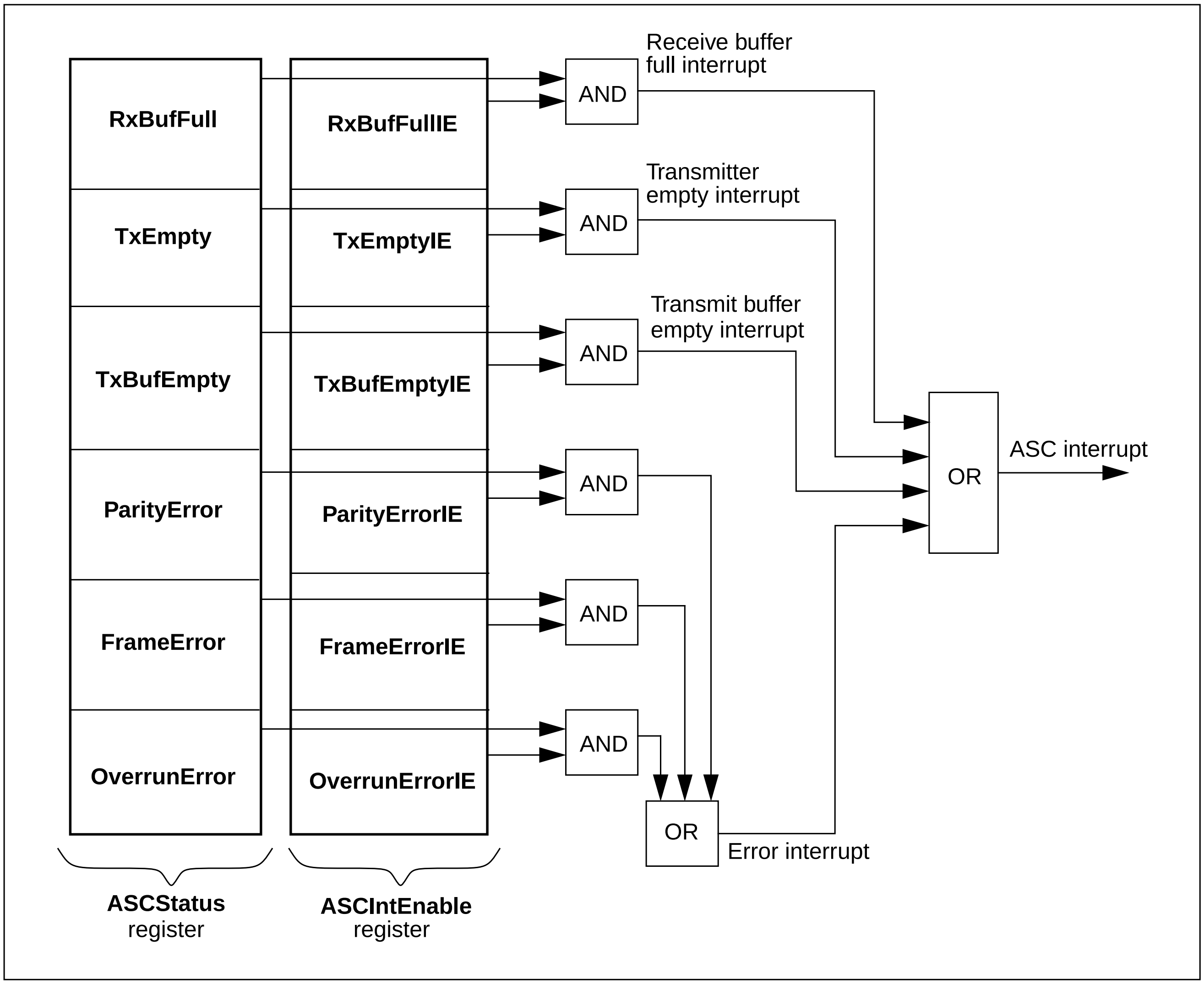


Figure 28.3 ASC status and interrupt registers

### 28.5.1 Using the ASC interrupts

The transmitter generates two interrupts; this provides advantages for the servicing software. For normal operation (i.e. other than the error interrupt) the ASC provides three interrupt requests to control data exchange via the serial channel:

- **TxBufEmpty TxHalfEmpty** is activated when data is moved from **ASC***n***TxBuffer** to the transmit shift register;
- **TxEmpty** is activated before the last bit of a frame is transmitted;
- **RxBufFull** is activated when the received frame is moved to **ASC***n***RxBuffer**.

As shown in Figure 28.4, **TxBufEmpty** is an early trigger for the reload routine, while **TxEmpty** indicates the completed transmission of the data field of the frame. Therefore, software using handshake should rely on **TxEmpty** at the end of a data block to make sure that all data has really been transmitted.

For single transfers it is sufficient to use the transmitter interrupt (**TxEmpty**), which indicates that the previously loaded data has been transmitted, except for the last bit of a frame.

For multiple back-to-back transfers it is necessary to load the next data before the last bit of the previous frame has been transmitted. The use of **TxEmpty** alone would leave just one stop bit time for the handler to respond to the interrupt and intiate another transmission. Using the output buffer interrupt (**TxBufEmpty**) to signal for more data allows the service routine to load a complete frame, as **ASC*n*TxBuffer** may be reloaded while the previous data is still being transmitted.

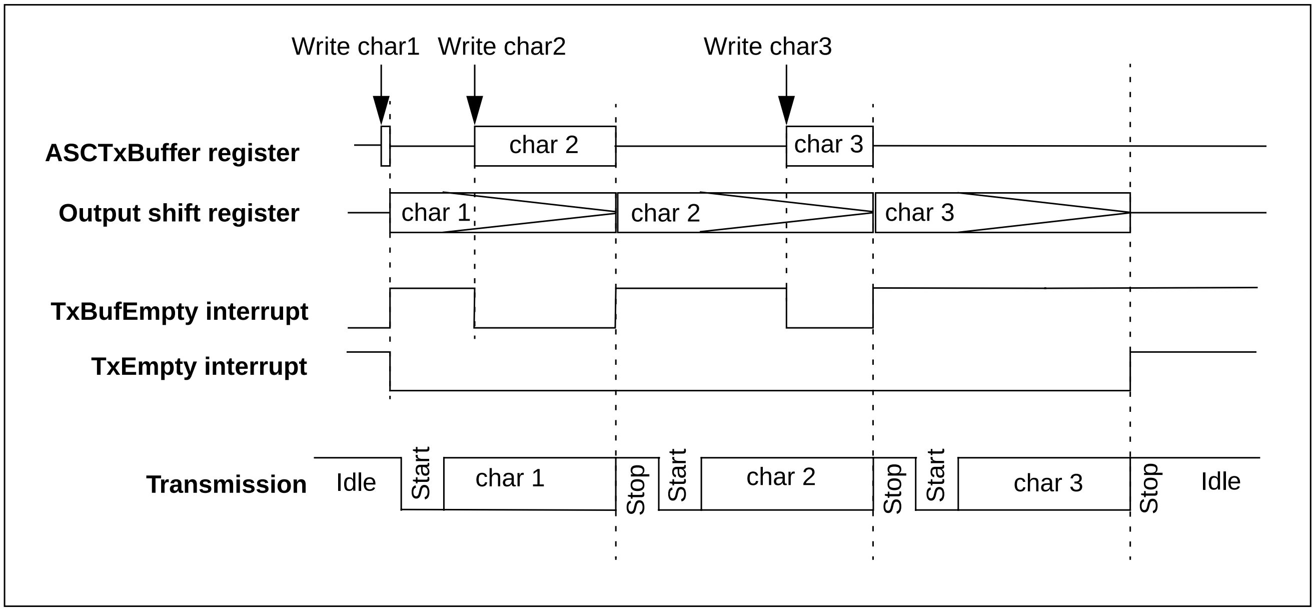


Figure 28.4 ASC transmission

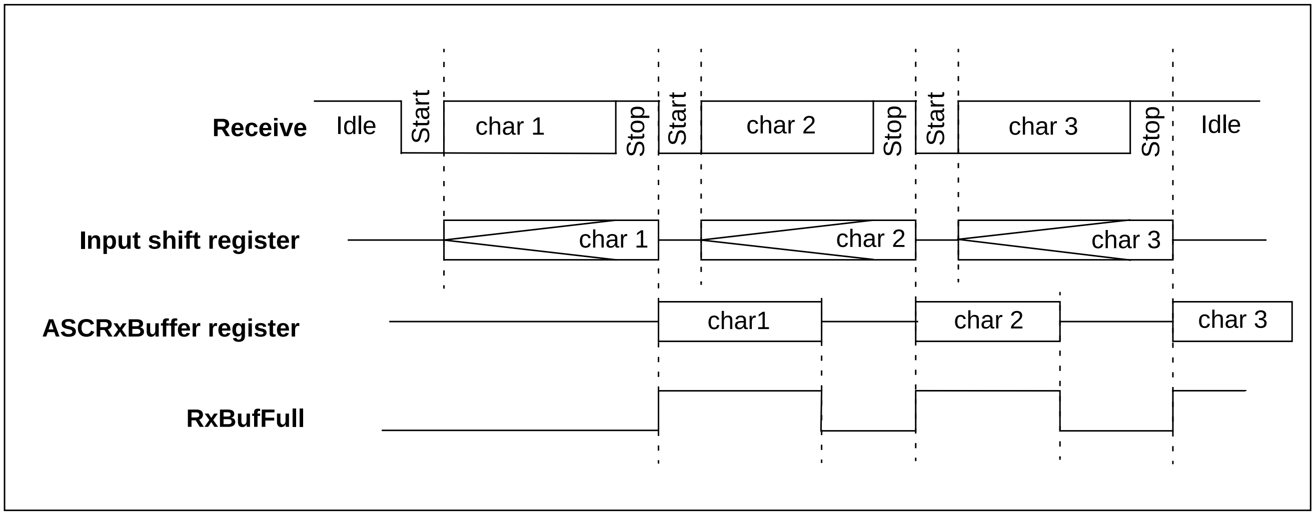


Figure 28.5 ASC receive

## 28.6 SmartCard mode specific operation

To conform to the ISO SmartCard specification the following modes are supported in the ASC SmartCard mode.

When the SmartCard mode bit is set to 1, the following operation occurs.

- Transmission of data from the transmit shift register is guaranteed to be delayed by a minimum of 1/2 baud clock. In normal operation a full transmit shift register will start shifting on the next baud clock edge. In SmartCard mode this transmission is further delayed by a guaranteed 1/2 baud clock.
- If a parity error is detected during reception of a frame programmed with a 1/2 stop bit period, the transmit line is pulled low for a baud clock period after the completion of the receive frame, i.e. at the end of the 1/2 stop bit period. This is to indicate to the SmartCard that the data transmitted to the UART has not been correctly received.
- The assertion of the **TxEmpty** interrupt can be delayed by programming the **ASC*n*GuardTime** register, as described in Table 28.9. In normal operation, **TxEmpty** is asserted when the transmit shift register is empty and no further transmit requests are outstanding.

  In SmartCard mode an empty transmit shift register triggers the guardtime counter to count up to the programmed value in the **ASC*n*GuardTime** register. **TxEmpty** is forced low during this time. When the guard time counter reaches the programmed value **TxEmpty** is asserted high.

  The de-assertion of **TxEmpty** is unaffected by SmartCard mode.

The receiver enable bit is reset after a character has been received. This avoids the receiver detecting another start bit in the case of the smartcard driving the **RXD** line low until the UART driver software has dealt with the previous character.

When the SmartCard mode bit is set to 0, normal UART operation occurs.

### 28.6.1 Guard time

The **ASC*n*GuardTime** register enables the user to delay the assertion of the interrupt **TxEmpty** by a programmable number of baud clock ticks.

| **ASC*n*GuardTime** | | **ASC*n*BaseAddress + #18** | **Read/Write** |
|---|---|---|---|
| **Bit** | **Bit field** | **Function** | |
| 7:0 | **GuardTime** | Number of baud clocks to delay assertion of **TxEmpty**. | |
| 15:8 | | Reserved. Write 0. Read back 0. | |

Table 28.9 **ASC*n*GuardTime** register format

# 29 SmartCard interface

The SmartCard interface is designed to support only asynchronous protocol SmartCards as defined in the *ISO7816-3* standard. Limited support for synchronous SmartCards can be provided in software by using PIO bits to provide the clock, reset, and I/O functions on the interface to the card. Two SmartCard interfaces are supported on the STi5500.

The UART function of the SmartCard interface is provided by a UART (ASC). UART ASC0 can be used by SmartCard0 and ASC2 can be used by SmartCard1.

Each ASC used by a SmartCard interface must be configured as eight data bits plus parity, 0.5 or 1.5 stop bits, with SmartCard mode enabled. A 16-bit counter, the SmartCard clock generator, divides down either the CPU clock, or an external clock connected to a pin shared with a PIO bit, to provide the clock to the SmartCard. PIO bits in conjunction with software are used to provide the rest of the functions required to interface to the SmartCard. The inverse signalling convention, as defined in *ISO7816-3*, is handled in software, inverted data and most significant bit first. See Chapter 28 for details of the ASC and Chapter 31 for details of the PIO ports.

## 29.1 External interface

The signals required by the SmartCard are given in Table 29.1.

| Pin | Function |
|---|---|
| Clk | Clock for SmartCard. |
| I/O | Input or output serial data. Open drain drive at both ends. |
| RST | Reset to card. |
| Vcc | Supply voltage. |
| Vpp | Programming voltage. |

Table 29.1 SmartCard pins

The signals provided on the STi5500 are given in Table 29.2.

| Pin | In/Out | Function |
|---|---|---|
| ScClk | Out, open drain for 5V cards. | Clock for SmartCard. |
| ScClkGenExtClk | In. | External clock input to SmartCard clock divider. |
| ScDataOut | Out, open drain driver. | Serial data output. Open drain drive. |
| ScDataIn | In. | Serial data input. |
| ScRST | Out, open drain. | Reset to card. |
| ScCmdVcc | Out. | Supply voltage enable/disable. |
| ScCmdVpp | Out. | Programming voltage enable/disable. |
| ScDetect | In. | SmartCard detection. |

Table 29.2 SmartCard interface pins

The **ScRST**, **ScCmdVpp**, **ScCmdVcc**, and **ScDetect** signals are provided by pins of the PIO ports. Programming the PIO pins of the port for alternative function modes connects the ASC **TXD** data signal to the **ScDataOut** pin with the correct driver type and the clock generator to the **ScClk** pin. Details of the PIO pin assignments can be found in Table 3.13.

The ISO standard defines the bit times for the asynchronous protocol in terms of a time unit called an ETU which is related to the clock frequency received by the card. One bit time is of length one ETU.

The ASC transmitter output and receiver input need to be connected together externally. For the transmission of data from the STi5500 to the SmartCard, the ASC will need to be set up in SmartCard mode.

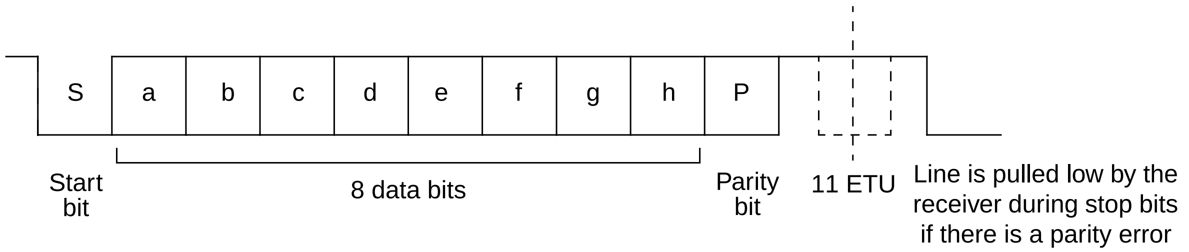

S
a
b
c
d
e
f
g
h
P
Start bit
8 data bits
Parity bit
11 ETU
Line is pulled low by the receiver during stop bits if there is a parity error

Figure 29.1 ISO 7816-3 asynchronous protocol

## 29.2 SmartCard clock generator

The SmartCard clock generator provides a clock signal to the connected SmartCard. The SmartCard uses this clock to derive the baud rate clock for the serial I/O between the SmartCard and another UART. The clock is also used for the CPU in the card, if present.

Operation of the SmartCard interface requires that the clock rate to the card is adjusted while the CPU in the card is running code, so that the baud rate can be changed or the performance of the card can be increased. The protocols that govern the negotiation of these clock rates and the altering of the clock rate are detailed in the *ISO7816-3* standard. The clock is used as the CPU clock for the SmartCard, so updates to the clock rate must be synchronized with the clock (**Clk**) to the SmartCard. This means the clock high or low pulse widths must not be shorter than either the old or new programmed value.

The clock generator clock source can be set to be either the system clock ( st20 processor clock ) or an external clock ( mpeg/smart card clock ) . Two registers control the period of the clock and the running of the clock.

The **ScClkVal** register determines the SmartCard clock frequency. The value given in the register is multiplied by 2 to give the division factor of the input clock frequency. The divider is updated with the new value for the divider ratio on the next rising or falling edge of the output clock.

The **ScClkCon** register controls the source of the clock and determines whether the SmartCard clock output is enabled. The programmable divider and the output are reset when the enable bit is set to 0.

# 30 I$^2$C interface (SSC)

The High-Speed Synchronous Serial Controller (SSC) can be used to interface to a wide variety of serial memories, remote control receivers, and other microcontrollers. Various interface standards exist for these, the most important of which is the I$^2$C bus in the set-top box application as this is the interface used most often for the control of the Link-IC. Figure 30.1 below shows how the SSC is interfaced to an I$^2$C bus as the bus master. Software is required to handle some of the I$^2$C bus protocol such as byte acknowledgement.

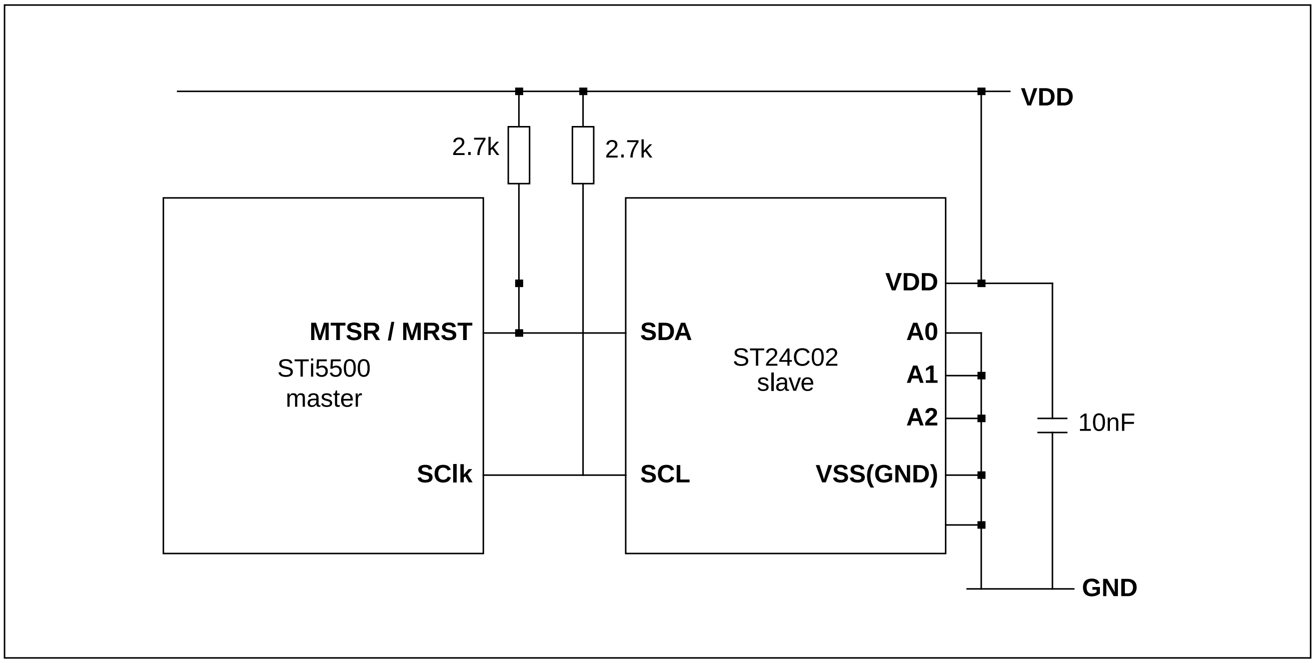


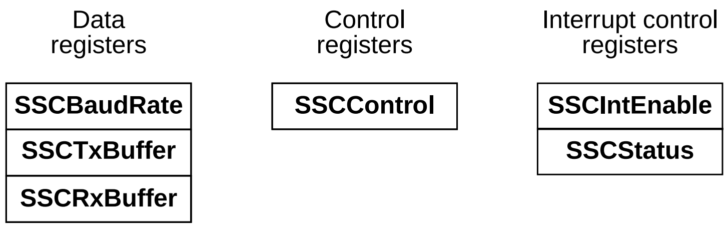
Figure 30.1 Connection of ST24C02 and STi5500 to I$^2$C-bus

The SSC provides flexible high-speed serial communication between the STi5500 and other microcontrollers, microprocessors or external peripherals using the I$^2$C bus protocol *as a master only*.

## 30.1 High-speed synchronous serial controller

The SSC supports half-duplex synchronous communication. The serial clock signal can be generated by the SSC itself (master mode). Data width is programmable. Transmission and reception of data is double-buffered. A 16-bit baud rate generator provides the SSC with a separate serial clock signal.

The high-speed synchronous serial controller can be used to communicate with shift registers (IO expansion), peripherals (e.g. EEPROMs) or other controllers (networking). The SSC supports half-duplex communication.



| | |
|---|---|
| **SSCBaudRate** | SSC Baud rate generator/reload register |
| **SSCTxBuffer** | SSC Transmit buffer register (write only) |
| **SSCRxBuffer** | SSC Receive buffer register (read only) |
| **SSCControl** | SSC Control register |
| **SSCIntEnable** | SSC Interrupt enable register |
| **SSCStatus** | SSC Status register |

Figure 30.2 Registers associated with the SSC

### Control register

The operating mode of the serial channel SSC is controlled by the control register (**SSCControl**).

| **SSCControl** | **SSC base address +#0C** | **Read/Write** |
|---|---|---|
| **Bit** | **Bit field** | **Function** |
| 3:0 | DataWidth | SSC Data width selection.<br>**DataWidth** **Meaning**<br>0000 Reserved. Do not use this combination.<br>0001 2 bits<br>0010 3 bits<br>... ...<br>1111 16 bits |
| 4 | HeadControl | SSC Heading control bit. For $I^2C$ operation, software *must* write a 1; the effect of writing 0 is undefined. The MSB of the selected data width is shifted out first. |
| 5 | ClkPhase | SSC Clock phase control bit. For $I^2C$ operation, software *must* write a 1; the effect of writing 0 is undefined |
| 6 | ClkPolarity | SSC Clock polarity control bit. For $I^2C$ operation, software *must* write a 0; the effect of writing 1 is undefined |
| 8 | MasterSel | SSC Master select bit. For $I^2C$ operation, software *must* write a 1; the effect of writing 0 is undefined |
| 9 | Enable | SSC Enable bit.<br>0 Transmission and reception disabled<br>1 Transmission and reception enabled |
| 10 | LoopBack | SSC Loopback bit.<br>0 transmitter is connected to shift register input<br>1 shift register output is connected to shift register input |
| 7, 15:11 | | RESERVED. Write 0, read back 0. |

Table 30.1 **SSCControl** register format

### 30.1.1 Synchronous serial channel operation

The shift register of the SSC is connected to both the transmit pin and the receive pin via the pin control logic (see block diagram Figure 30.3). Transmission and reception of serial data is synchronized and takes place at the same time, i.e. the same number of transmitted bits is also received. Transmit data is written into the Transmit Buffer (**SSCTxBuffer**) register. It is moved to the shift register as soon as this is empty. The SSC immediately begins transmitting. When the data has transferred to the shift register, the transmit buffer empty (**TxBufEmpty**) flag will be set to indicate that the transmit buffer (**SSCTxBuffer**) may be reloaded again. When the programmed number of bits (2 to 16) has been transferred, the contents of the shift register are moved to the Receive Buffer (**SSCRxBuffer**) register and the receive buffer full (**RxBufFull**) flag will be set. If no further transfer is to take place, i.e. the transmit buffer is empty, the SSC will revert back to an idle state waiting for a load of the transmit register.

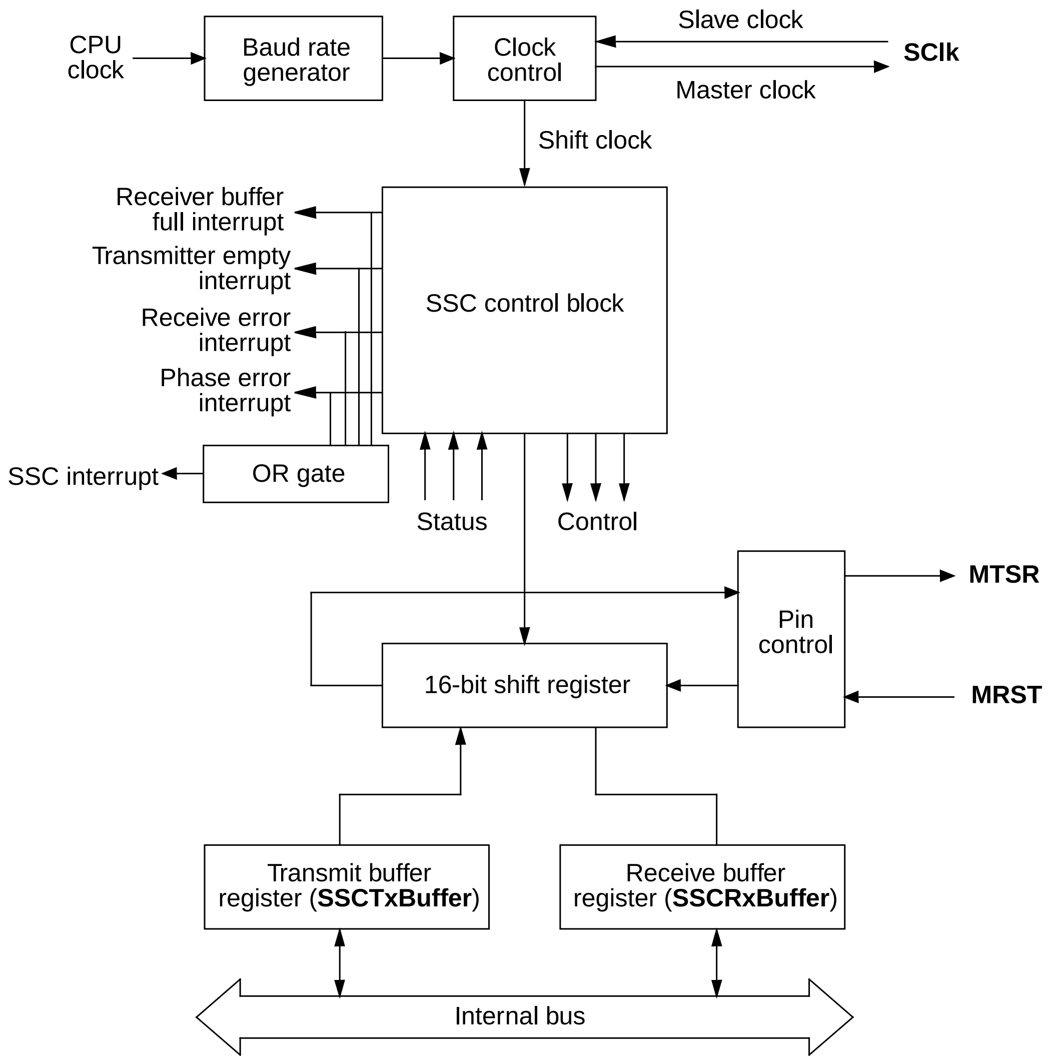


Figure 30.3 Synchronous serial channel SSC block diagram

Note that only one SSC can be master at a given time.

The transfer of serial data bits can be programmed as follows:

- the data width can be 2 to 16 bits
- the baud rate can be set over a wide range

The data width selection (**DataWidth**) bit allows data widths of 2 to 16 bits to be transferred.

The unused bits of **SSCTxBuffer** are ignored, the unused bits of **SSCRxBuffer** are not valid and should be ignored by the receiver service routine.

### Transmit and receive buffer registers

| **SSCTxBuffer** | **SSC base address + #04** | **Write only** |
|---|---|---|
| **Bit** | **Bit field** | **Function** |
| 15:0 | TD15:0 | Transmit buffer data **D15:0** |

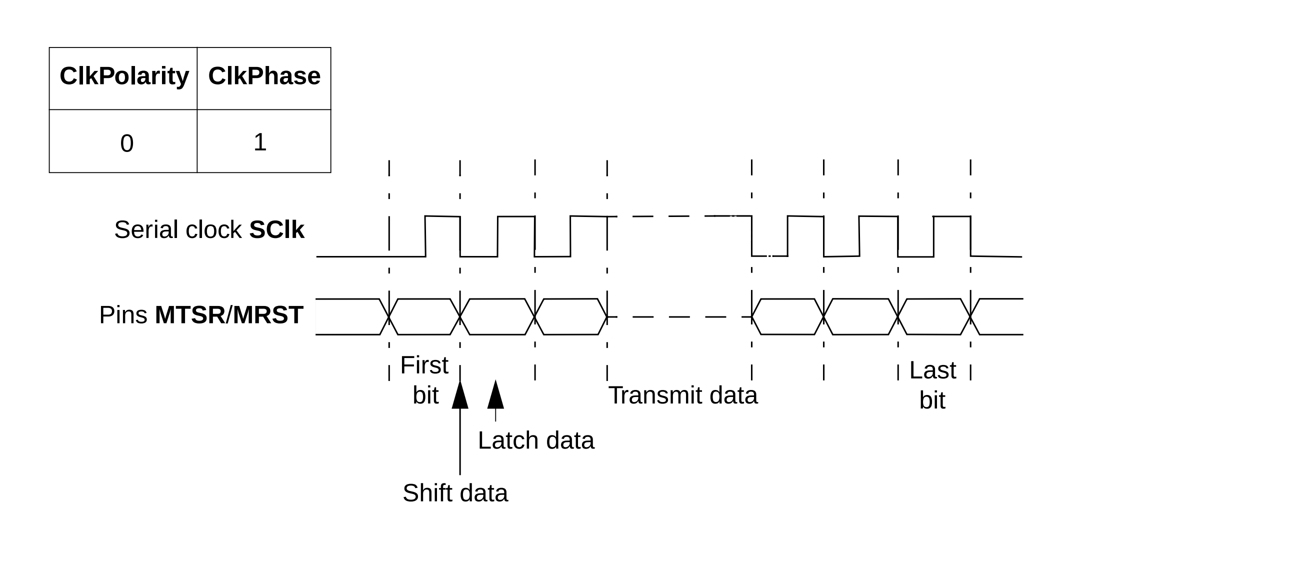
Table 30.2 **SSCTxBuffer** register format

| **SSCRxBuffer** | **SSC base address + #08** | **Read only** |
|---|---|---|
| **Bit** | **Bit field** | **Function** |
| 15:0 | RD15:0 | Receive buffer data **D15:0** |

Table 30.3 **SSCRxBuffer** register format

### Clock control

If the **ClkPhase** and **ClkPolarity** bits in the **SSCControl** register are programmed, as defined by Table 30.1 on page 257, then the clock and data relationship will be I$^2$C compatible. The data is stable during the high level of the clock and I$^2$C setup and hold times are met.



| **ClkPolarity** | **ClkPhase** |
|---|---|
| 0 | 1 |

Figure 30.4 Clock and data relationships

### 30.1.2 Half-duplex operation

In a half duplex configuration only one data line is necessary for both receiving *and* transmitting of data. The data exchange line is connected to both pins **MTSR** and **MRST** of each device, the clock line is connected to the **SClk** pin.

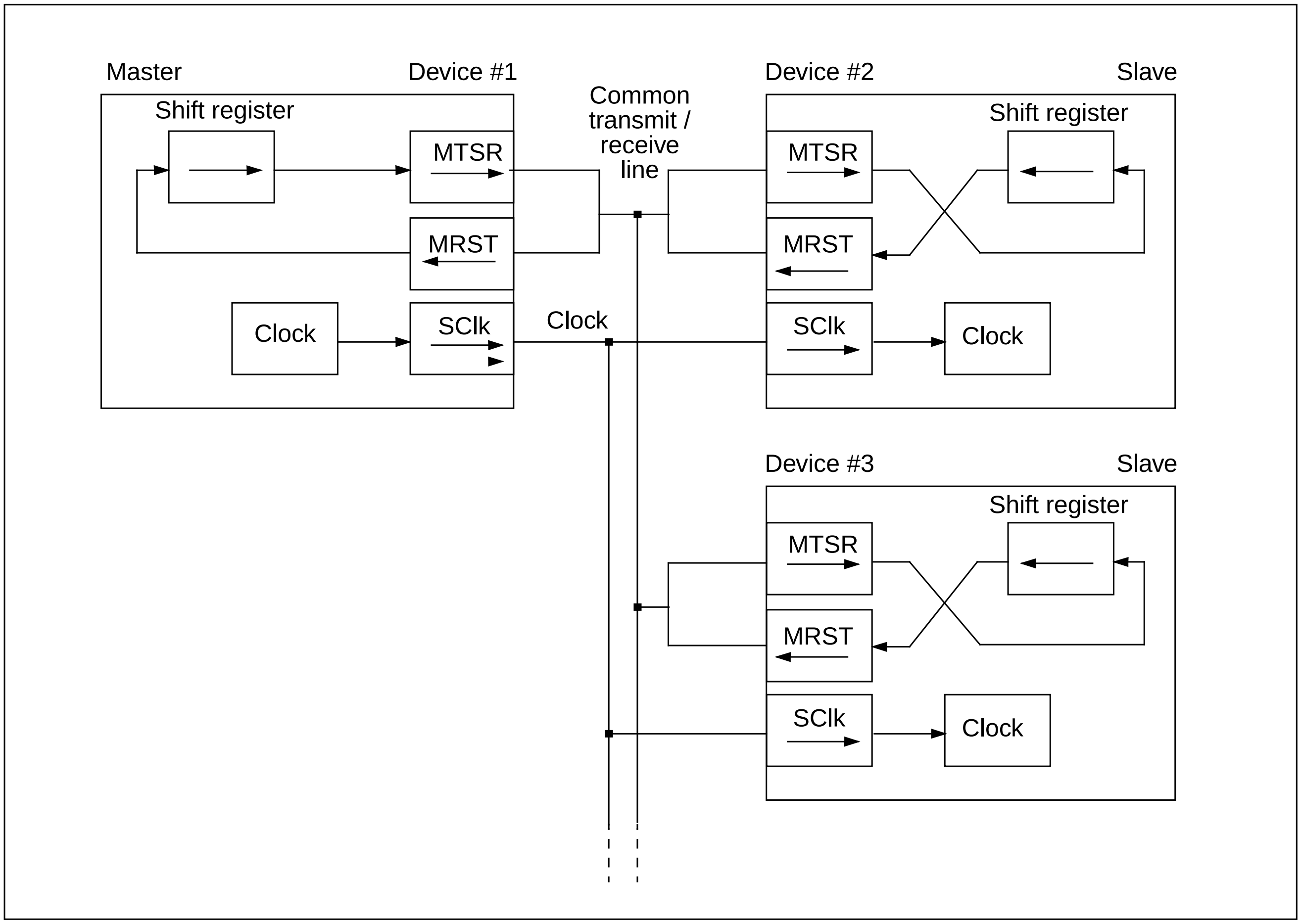


Figure 30.5 Half-duplex configuration

The master device controls the data transfer by generating the shift clock, while the slave devices receive it. Due to the fact that all transmit and receive pins are connected to the one data exchange line, serial data may be moved between arbitrary stations.

Similar to full duplex mode there are *two ways to avoid collisions* on the data exchange line:

- only the transmitting device may enable its transmit pin driver
- the non-transmitting devices use open drain output and only send ones.

Since the data inputs and outputs are connected together, a transmitting device will clock its own data at the input pin (**MRST** for a master device). This allows any corruptions on the common data exchange line, where the received data is not equal to the transmitted data, to be detected.

### Continuous transfers

When the **TxBufEmpty** bit is 1, it indicates that the transmit buffer **SSCTxBuffer** is empty and ready to be loaded with the next transmit data. If **SSCTxBuffer** has been reloaded by the time the current transmission is finished, the data is immediately transferred to the shift register and the next transmission will start without any additional delay. On the data line there is no gap between the two successive frames. For example, two byte transfers would look the same as one word transfer. This feature can be used to interface with devices which can operate with or require more than 16 data bits per transfer. Software determines how long a total data frame length can be. This option can also be used to interface to byte-wide and word-wide devices on the same serial bus.

Note: This can only happen in multiples of the selected basic data width, since it would require disabling/enabling of the SSC to reprogram the basic data width on-the-fly.

## 30.1.3 Baud rate generation

The SSC has its own dedicated 16-bit baud rate generator with 16-bit reload capability. The resultant baud rate for transmission and reception is *half* the value in the **SSCBaudRate** register.

## 30.1.4 Baud rate generator register

| **SSCBaudRate** | | **ASC base address + #00** | **Read/Write** |
|---|---|---|---|
| **Bit** | **Bit field** | **Write Function** | **Read Function** |
| 15:0 | ReloadVal | 16-bit reload value | 16-bit count value |

Table 30.4 **SSCBaudRate** register format

### Baud rates

The formulas below calculate either the resulting baud rate for a given reload value, or the required reload value for a given baud rate:

$$\text{Baudrate} = \frac{f_{CPU}}{2 \text{ x } \text{<SSCBaudRate>}} \qquad \text{<SSCBaudRate>} = \left(\frac{f_{CPU}}{2 \text{ x Baudrate}}\right)$$

Where, <SSCBaudRate> represents the content of the reload register, as an unsigned 16-bit integer and $f_{CPU}$ represents the CPU clock frequency.

The maximum baud rate that can be achieved when using a CPU clock of 40 MHz is 5 MBaud. Table 30.5 below lists some possible baud rates together with the required reload values and the resulting bit times, assuming a CPU clock of 40 MHz.

| Baud rate | Bit time | Reload value |
|---|---|---|
| Reserved. Use a reload value > 0. | - | #0000 |
| 5 MBaud | 200 ns | #0004 |
| 3.3 MBaud | 300 ns | #0006 |
| 2.5 MBaud | 400 ns | #0008 |
| 2.0 MBaud | 500 ns | #000A |
| 1.0 MBaud | 1 µs | #0014 |
| 100 KBaud | 10 µs | #00C8 |
| 10 KBaud | 100 µs | #07D0 |
| 1.0 KBaud | 1 ms | #4E20 |

Table 30.5 Baud rates and bit times for different **SSCBaudRate** reload values

Note: The content of **SSCBaudRate** must be greater than 0.

### 30.1.5 Hardware error detection capabilities

The SSC is able to detect two different error conditions.

- *Receive Error*
- *Phase Error*

When an error is detected, the respective error flag is set in the **SCCStatus** register. The error interrupt handler may then check the error flags to determine the cause of the error interrupt.

A *Receive Error* is detected, when a new data frame is completely received, but the previous data was not read out of the receive buffer register **SSCRxBuffer**. This condition sets the error (**RxError**) flag and, when enabled via **RxErrorIE,** the error interrupt request flag (**ErrorInterrupt**)**.** The old data in the receive buffer **SSCRxBuffer** will be overwritten with the new value and is irretrievably lost.

A *Phase Error* is detected, when the incoming data on the **MRST** pin, sampled at the same frequency as the CPU clock, changes between one sample before and two samples after the latching edge of the clock signal (See “Clock control” on page 259.). This condition sets the error flag **PhaseError** and, when enabled via **PhaseErrorIE**, the error interrupt request flag (**ErrorInterrupt**).

### 30.1.6 Interrupt control

The SSC contains two registers that are used to control interrupts, a status (**SSCStatus)** register and an interrupt enable (**SSCIntEnable**) register. The status bits in the **SSCStatus** register determine the cause of the interrupt. Interrupts will occur when a status bit is 1 (high) and the corresponding bit in the **SSCIntEnable** register is 1.

The error interrupt signal (**ErrorInterrupt**) is generated by the SSC from the OR of the receive error and phase error status bits after they have been ANDed with the corresponding enable bits in the **SSCIntEnable** register.

An overall interrupt request signal (**SSC_interrupt)** is generated from the OR of the receive interrupt request (**RxBufFull**), transmit interrupt request (**TxBufEmpty**) and error interrupt request (**ErrorInterrupt**) signals.

Note the status register *cannot* be written to directly by software. The set and reset mechanism for the status register is described below.

The receiver interrupt status bit (**RxBufFull**) is set when a character is loaded from the shift register into the receive buffer (**SSCRxBuffer**). The **RxBufFull** bit is reset when a character is read from the receive buffer (**SSCRxBuffer**).

The transmitter interrupt status bit (**TxBufEmpty**) is set when a character is loaded from the transmitter buffer (**SSCTxBuffer**) into the shift register. The **TxBufEmpty** bit is reset when a character is written into the transmitter buffer (**SSCTxBuffer**).

The status bits (**RxError, PhaseError**) are reset when a character is read from the receive buffer (**SSCRxBuffer**).

| **SSCStatus** | **SSC base address + #14** | **Read Only** |
|---|---|---|
| **Bit** | **Bit field** | **Function** |
| 0 | RxBufFull | Receiver Buffer Full Flag<br>1 receiver buffer full |
| 1 | TxBufEmpty | Transmitter Buffer Empty Flag<br>1 transmitter buffer empty |
| 3 | RxError | Receive Error Flag<br>1 receive error set |
| 4 | PhaseError | Phase Error Flag<br>1 phase error set |
| 2, 7:5 | | RESERVED. Will read back 0. |

Table 30.6 **SSCStatus** register format

| **SSCIntEnable** | **SSC base address + #10** | **Read/Write** |
|---|---|---|
| **Bit** | **Bit field** | **Function** |
| 0 | RxBufFullIE | Receiver Buffer Full Interrupt Enable<br>1 receiver buffer full interrupt enable |
| 1 | TxBufEmptyIE | Transmitter Buffer Empty Interrupt Enable<br>1 transmitter buffer empty interrupt enable |
| 3 | RxErrorIE | Receive Error Interrupt Enable<br>1 receive error interrupt enable |
| 4 | PhaseErrorIE | Phase Error Interrupt Enable<br>1 phase error interrupt enable |
| 2, 7:5 | | RESERVED. Write 0, will read back 0. |

Table 30.7 **SSCIntEnable** register format

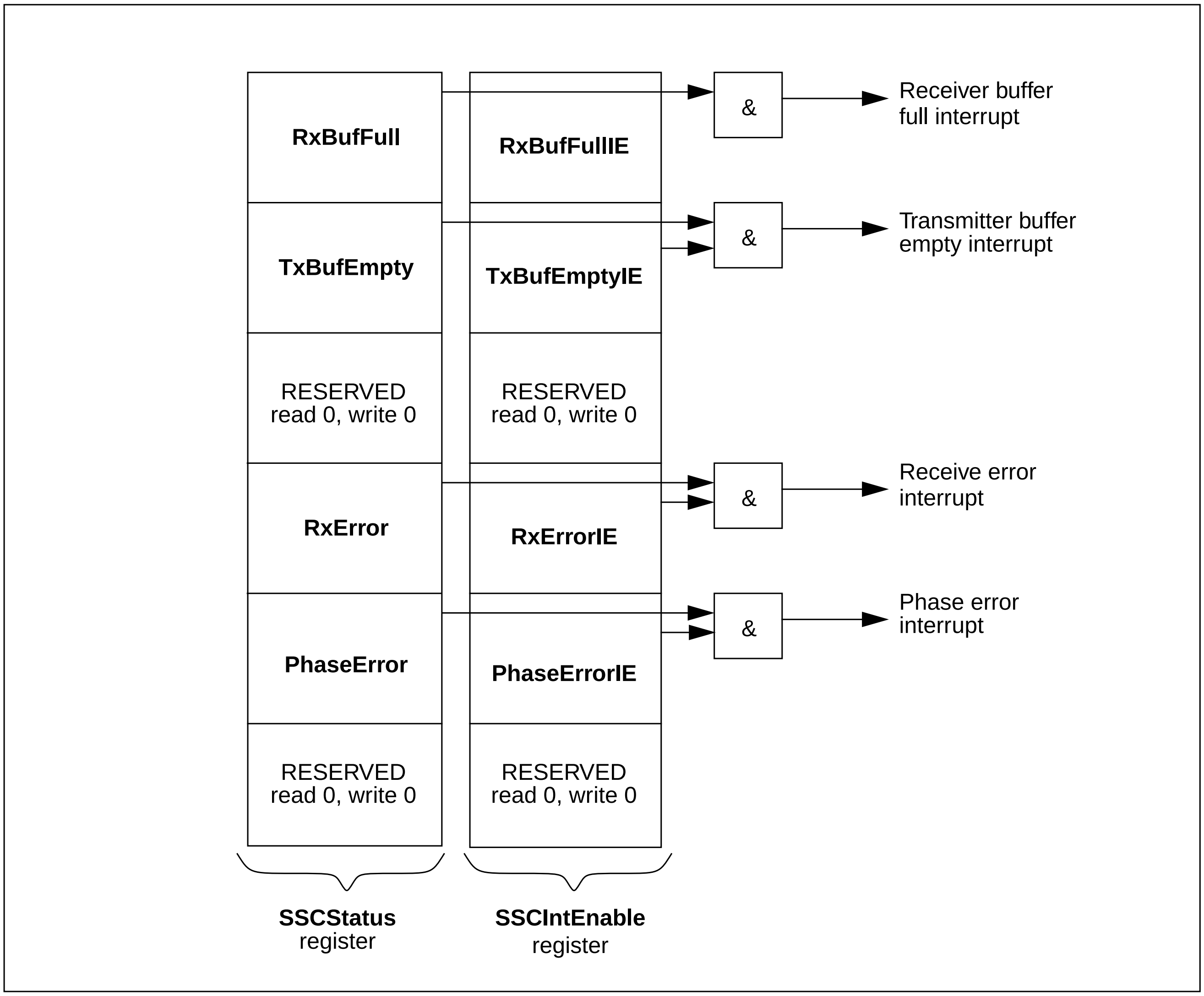


Figure 30.6 SSC status and interrupt registers

### Using the SSC interrupts

An interrupt handler for the SSC needs to read the **SCCStatus** register before writing the **SCCTx-Buffer** or reading the **SCCRxBuffer** as there might have been an error. The error flags will be cleared by these read or write operations, see sections above on error detection and interrupts.

# 31 Parallel input/output

The STi5500 device has 34 bits of parallel input/output (PIO), configured in four ports **PIO3-0** of eight bits and one port **PIO4** of two bits. Each bit is programmable as an output, an input, a bidirectional pin, or as an alternative function output pin. The alternative function connects signals from device peripherals to the pins of the device through the PIO. Details of the alternative function assignments can be found in Chapter 3. Not all of the PIO port bits are connected to device pins.

Each port of eight input bits can also be compared against a register and an interrupt generated when the value is not equal. Each of the ports operates as described in the rest of this chapter.

Output drivers for the PIO pins, both in PIO mode and the alternative function mode, can be programmed to be push-pull or open drain.

Each 8-bit PIO port has a set of 8-bit registers, and the 2-bit port has a similar set of 2-bit registers. Each of the bits of each register refers to the corresponding pin in the corresponding port. These registers hold:

- the output data for the port (**P***n***Out**);
- the input data read from the pin (**P***n***In**);
- PIO bit configuration registers (**P***n***C0-2**);
- the two input compare function registers (**P***n***Comp** and **P***n***Mask**).

Each of the registers, except **P***n***In**, is mapped onto two additional addresses so that bits can be set or cleared individually. The **Set_** register allows bits to be set individually. Writing a ‘1’ in this register sets the corresponding bit in the associated register, a ‘0’ leaves the bit unchanged. Similarly the **Clear_** register allows bits to be cleared individually. Writing a ‘1’ in this register resets the corresponding bit in the associated register; a ‘0’ leaves the bit unchanged.

# Part F Timing and electrical data

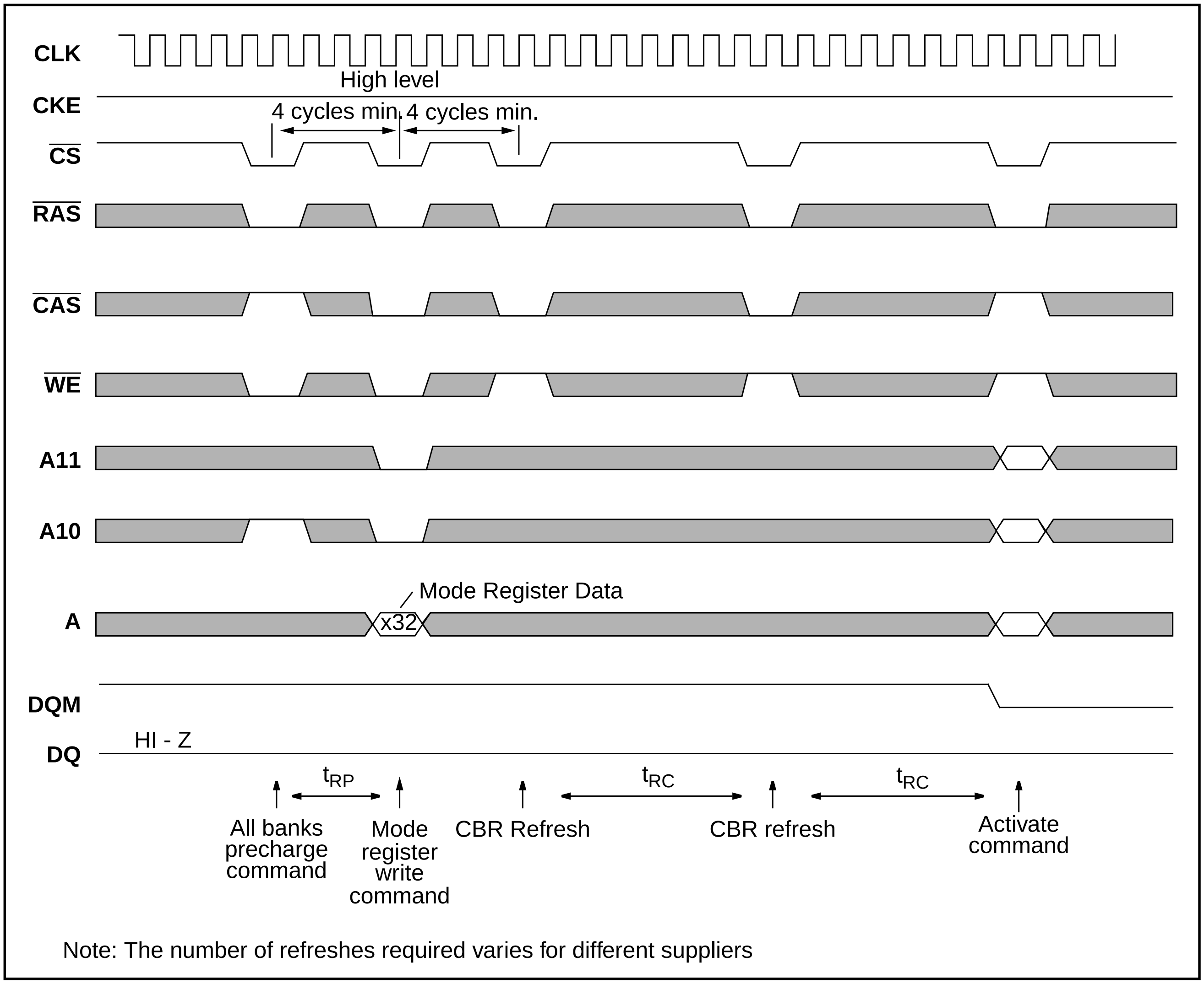
# 32 Timing specifications

The STi5500 is being characterized. These timing specifications are given for guidance only and will be updated when characterization is complete.

The timings are based on the following conditions unless otherwise stated:

1. Input rise and fall times of 3 ns (10% -> 90%).
2. Output load = 30pF
3. Output threshold = 1.5V

## 32.1 SDRAM



Note: The number of refreshes required varies for different suppliers

Figure 32.1 Synchronous DRAM power-on sequence

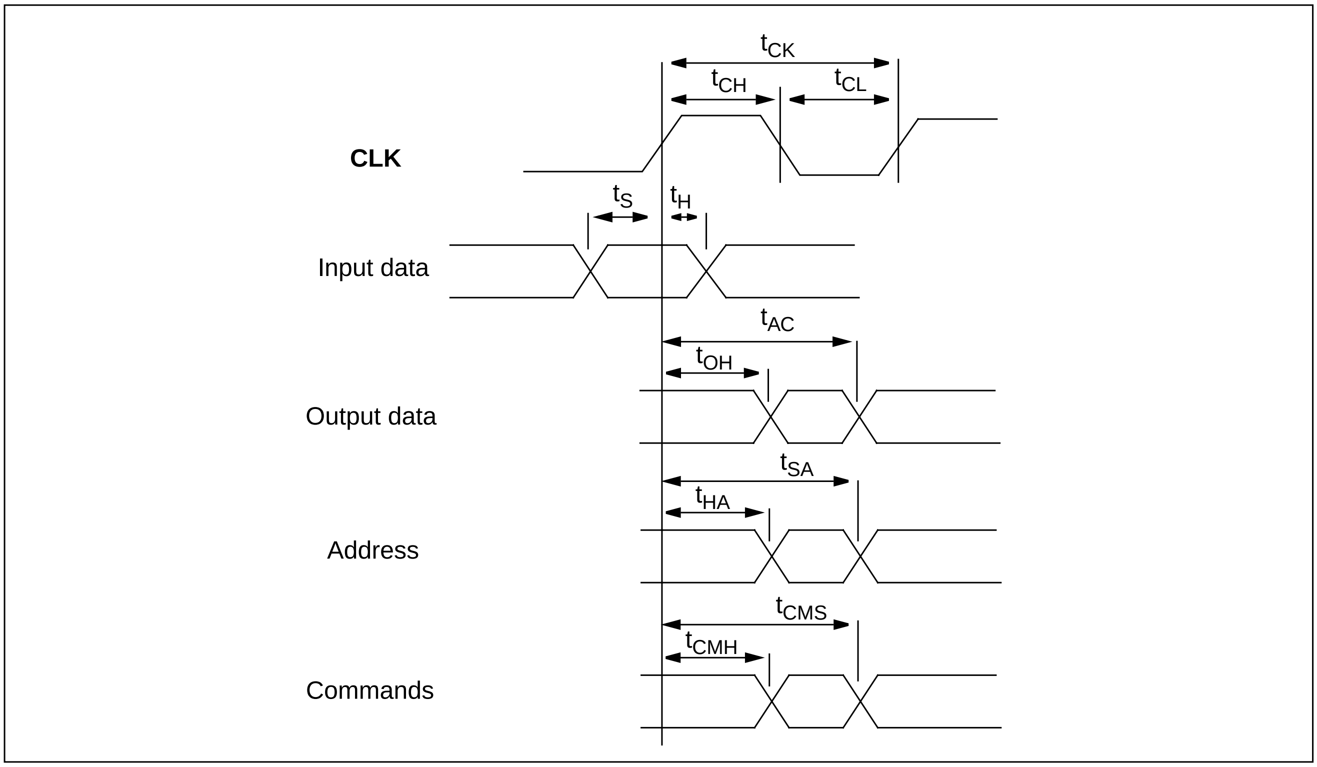


Figure 32.2 AC parameters for read and write (synchronous DRAM)

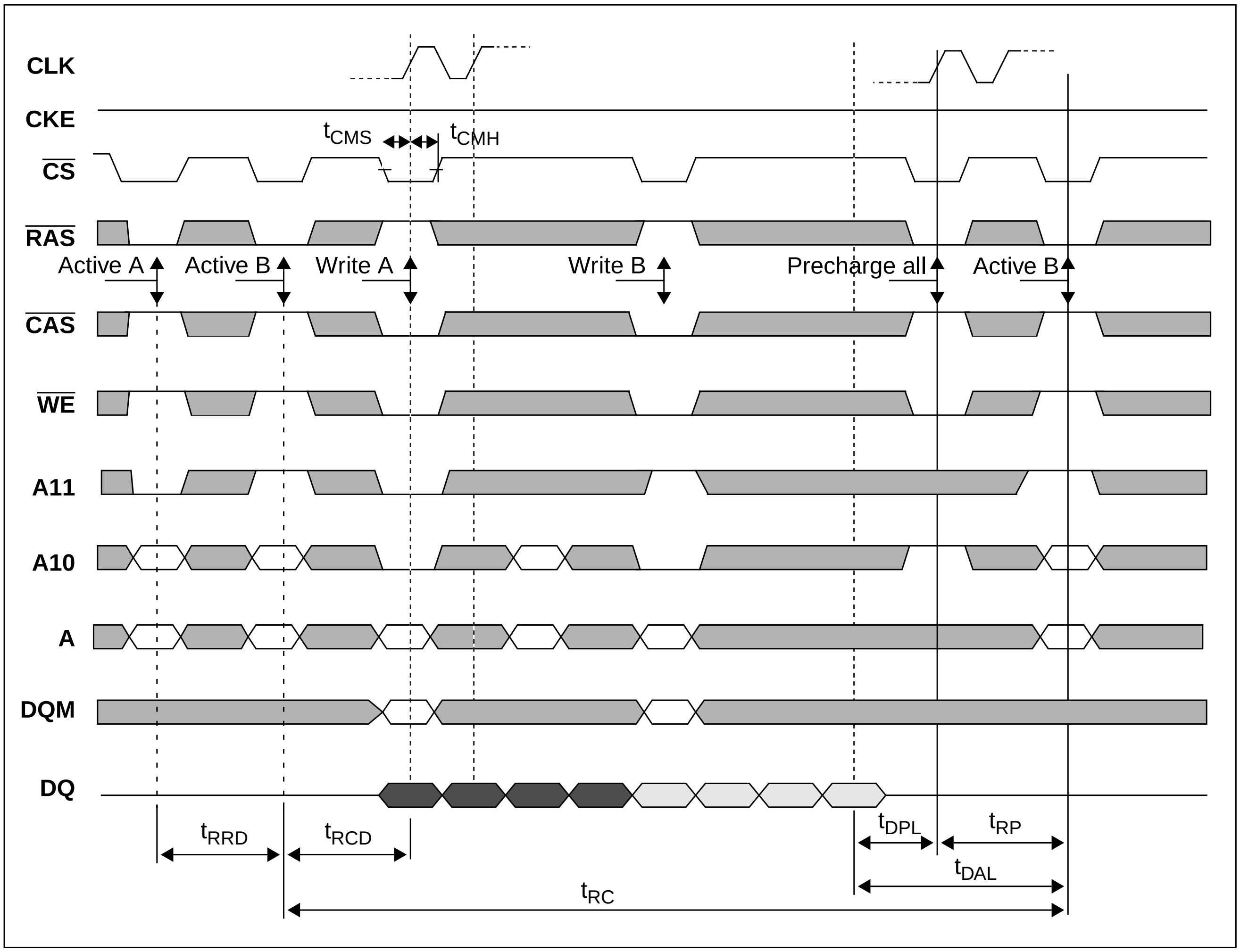


Figure 32.3 Synchronous DRAM write (burst length = 4, CAS latency = 3)

In Table 32.1, the unit T is the period of the memory subsystem clock, which is typically 100MHz.

| Symbol | Parameter | Min | Max | Units | Notes |
|---|---|---|---|---|---|
| $t_{RP}$ | ACTIVE to PRE Command Period | 3 | | T | |
| $t_{RC}$ | REF to REF / ACTIVE Command Period | 8 | | T | |
| $t_{CK}$ | Clock Cycle time | 9.25 | | ns | 1 |
| $t_{CH}$ | Clock HIGH level width | | | ns | |
| $t_{CL}$ | Clock LOW level width | | | ns | |
| $t_S$ | Data input setup time | 0 | | ns | |
| $t_H$ | Data input hold time | 2.5 | | ns | |
| $t_{AC}$ | Output data access time | | 2.9 | ns | |

Table 32.1  Synchronous DRAM read and write

| Symbol | Parameter | Min | Max | Units | Notes |
|---|---|---|---|---|---|
| $t_{OH}$ | Output data hold time | 1 | | ns | |
| $t_{SA}$ | Address access time | | 7 | ns | |
| $t_{HA}$ | Address hold time | 1.7 | | ns | |
| $t_{CMS}$ | Command ($\overline{CS}$, $\overline{RAS}$, $\overline{CAS}$, $\overline{WE}$,DQM) access time | | 2.7 | ns | |
| $t_{CMH}$ | Command ($\overline{CS}$, $\overline{RAS}$, $\overline{CAS}$, $\overline{WE}$,DQM) hold time | 1 | | ns | |
| $t_{RCD}$ | Delay time ACTIVE to READ / WRITE command | 4 | | T | |
| $t_{RRD}$ | ACTIVE(A) to ACTIVE(B) Command period | 4 | | T | |
| $t_{DAL}$ | Data-out to ACTIVE command period | 5 | | T | |
| $t_{DPL}$ | Data-out to Precharge Command period | 2 | | T | |
| $t_{RAS}$ | ACTIVE to PRECHARGE Command Period | 9 | | T | |

Table 32.1 Synchronous DRAM read and write

1 A 50% duty cycle is only obtained using the hard wired divide by 2, which is recommended.

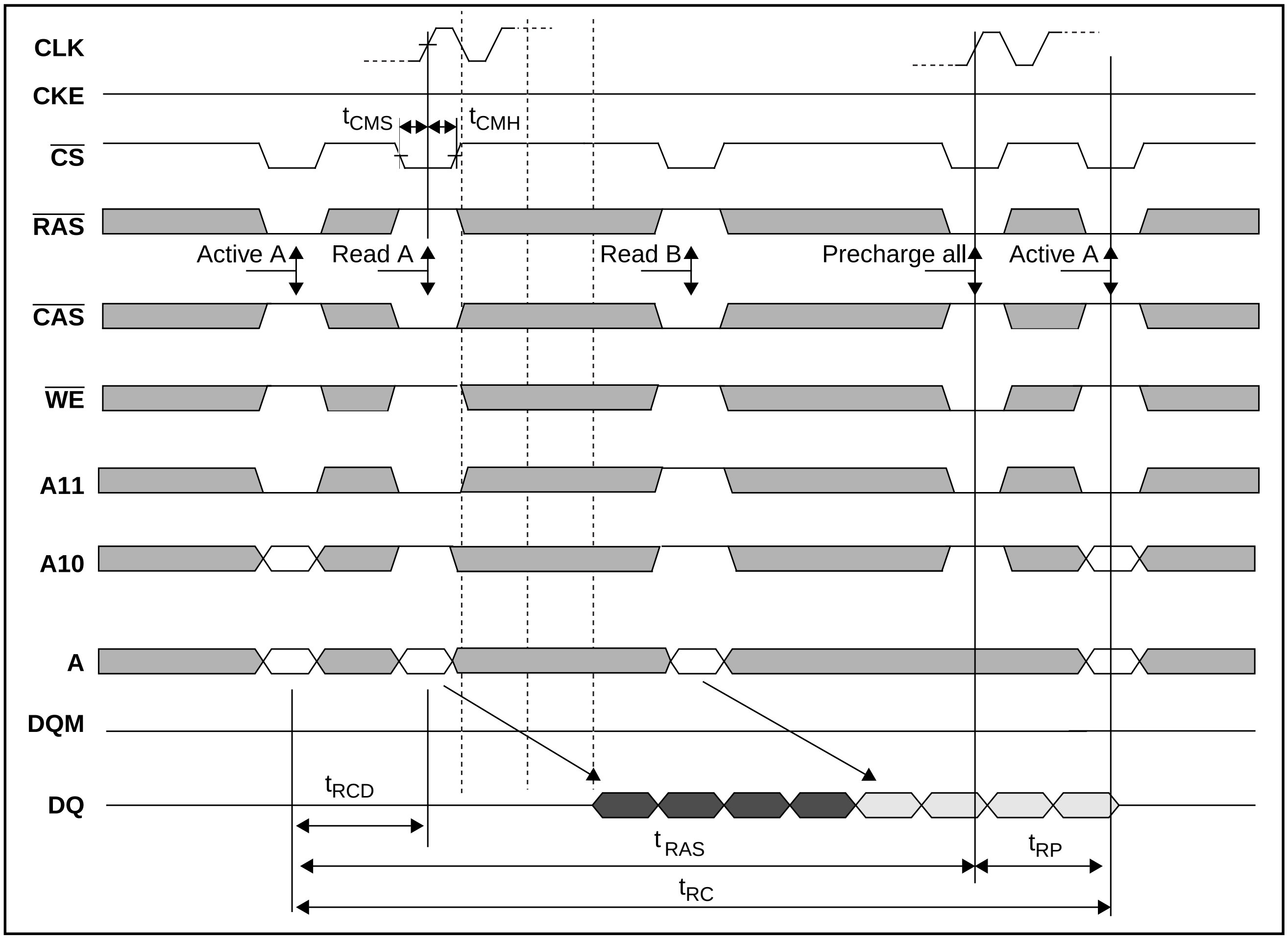


Figure 32.4 Synchronous DRAM read (burst length = 4, CAS latency = 3)

## 32.2 PCM/AC-3 decoder interface

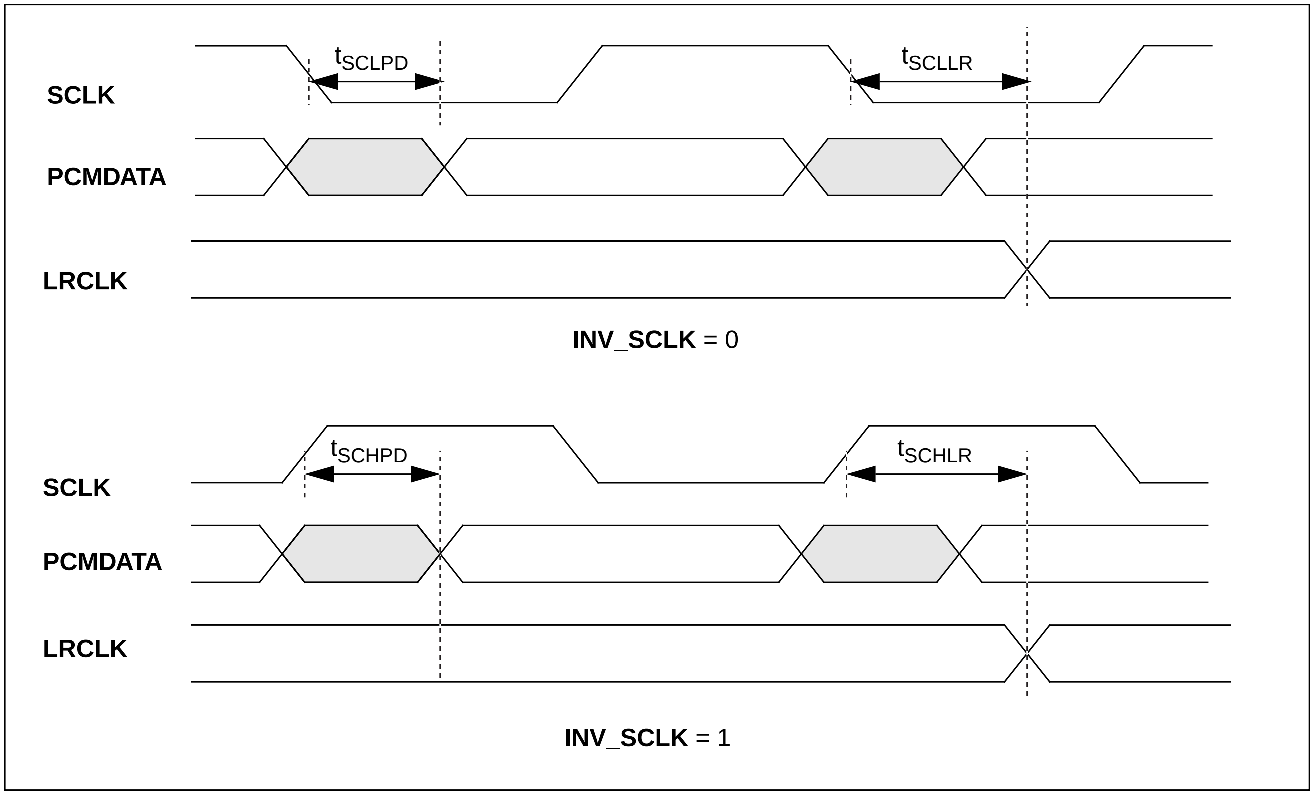


Figure 32.5 PCM data output

| Symbol | Parameter | Min | Max | Units | Notes |
|---|---|---|---|---|---|
| $t_{SCLPD}$ | SC low to PCMDATA valid | | 50 | ns | |
| $t_{SCLLR}$ | SC low to LRCLK | | 50 | ns | |
| $t_{SCLPD}$ | SC low to PCMDATA valid | | 50 | ns | |
| $t_{SCLLR}$ | SC low to LRCLK | | 50 | ns | |

Table 32.2  PCM data output

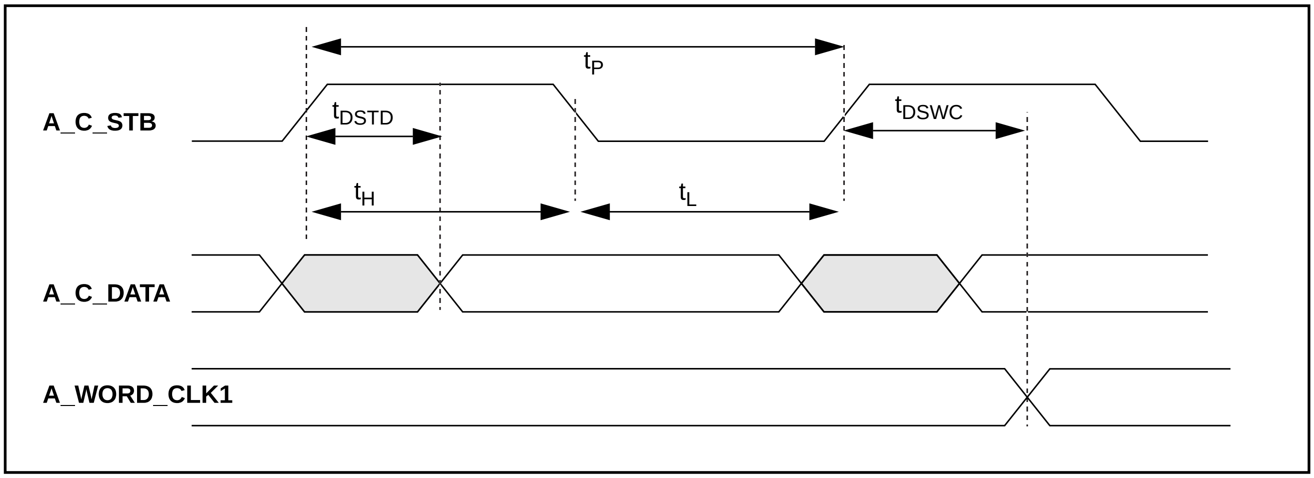


Figure 32.6 AC-3 decoder interface

| Symbol | Parameter | Min | Max | Units | Notes |
|---|---|---|---|---|---|
| $t_{DSTD}$ | Delay strobe to data | 30 | 40 | ns | |
| $t_{DSWC}$ | Delay strobe to word clock | 30 | 40 | ns | |
| $t_P$ | Strobe period | 70 | | ns | |
| $t_H$ | Strobe pulse width, high | 30 | | ns | |
| $t_L$ | Strobe pulse width, low | 30 | | ns | |

Table 32.3 AC-3 decoder interface

## 32.3 EMI timings

The *EMI Reference Clock* used in the EMI timings is a virtual clock and is defined as the point at which all positively edged EMI strobe and address outputs are valid. This removes process dependent skews from the datasheet description and highlights the dominant influence of address and strobe timings on memory system design.

| Symbol | Parameter | Min | Max | Units | Note |
|---|---|---|---|---|---|
| tCHAV | Reference Clock high to Address valid | -8.0 | 0.0 | ns | |
| tCLSV | Reference Clock low to Strobe valid | -8.0 | 3.0 | ns | |
| tCHSV | Reference Clock high to Strobe valid | -8.0 | 0.0 | ns | |
| tRDVCH | Read Data valid to Reference Clock high | 13.0 | | ns | |
| tCHRDX | Read Data hold after Reference Clock high | | -2.0 | ns | |
| tSVRDX | Read Data hold after Strobe valid | 0.0 | | ns | 1 |
| tCLWDV | Reference Clock low to Write Data valid | -8.0 | 7.0 | ns | 1 |
| tCHWDV | Reference Clock high to Write Data valid | -8.0 | 6.0 | ns | 1 |

Table 32.4 EMI cycle timings

| Symbol | Parameter | Min | Max | Units | Note |
|---|---|---|---|---|---|
| $t_{CHWDZ}$ | Reference Clock high to write data tristate | -8.0 | 6.0 | ns | |
| $t_{CHRSV}$ | Reference Clock high to remaining Strobes valid | -8.0 | 3.0 | ns | |
| $t_{CHPH}$ | Reference Clock high to **ProcClkOut** high | -8.0 | 0.0 | ns | |
| $t_{WVCH}$ | **MemWait** valid to Reference Clock high | 13.0 | | ns | |
| $t_{RVCH}$ | **MemReq** valid to Reference Clock high | 13.0 | | ns | |
| $t_{PHWX}$ | **MemWait** hold after **ProcClkOut** high | 0.0 | | ns | 1 |
| $t_{PHRX}$ | **MemReq** hold after **ProcClkOut** high | 0.0 | | ns | 1 |
| $t_{PHEMIZ}$ | **MemGrant** to signals tristate when bus granted | TBD | | ns | 1 |

Table 32.4  EMI cycle timings

### Notes

1 Minimum values are guaranteed by design.

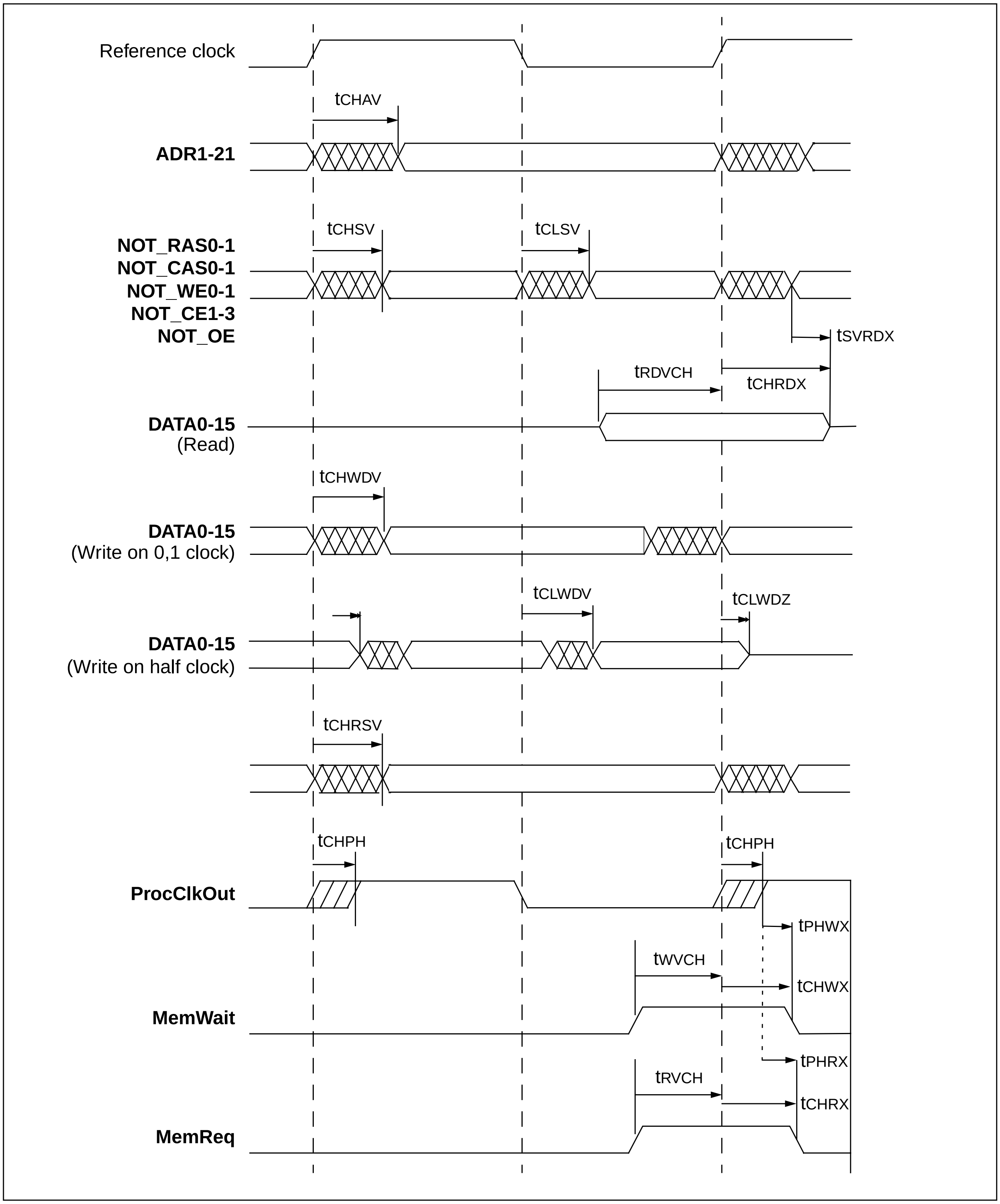


Figure 32.7 EMI timings

## 32.4 Rise and fall times

All rise and fall times are measured at 10 – 90%.

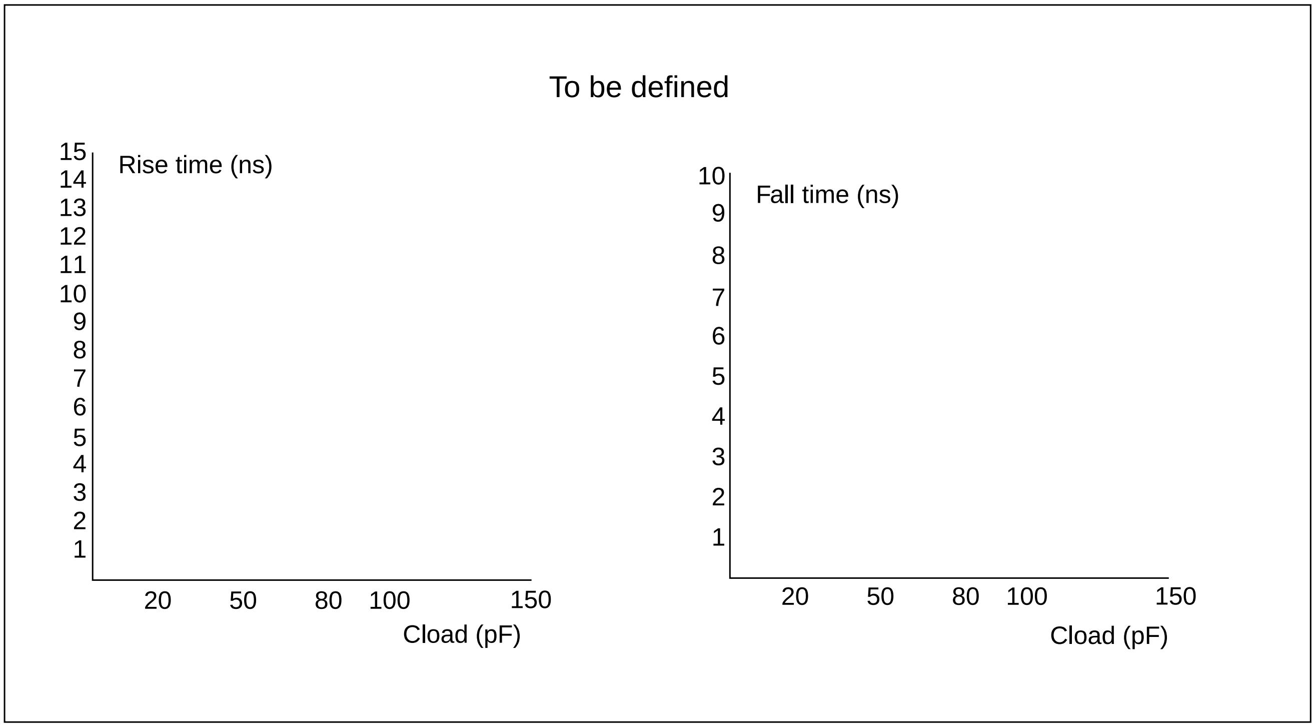


Figure 32.8 Rise and fall times for EMI pins

## 32.5 PIO timings

Reference clock in this case means the last transition of any PIO signal.

| Symbol | Parameter | Min | Max | Units | Note |
|---|---|---|---|---|---|
| $t_{PCHPOV}$ | **PIO_refclock** high to PIO output valid | -15.0 | 0.0 | ns | |
| $t_{PCHWDZ}$ | PIO tristate after **PIO_refclock** high | -15.0 | 5.0 | ns | |
| $t_{PIOr}$ | Output rise time | 7.0 | 30.0 | ns | |
| $t_{PIOf}$ | Output fall time | 7.0 | 30.0 | ns | |

Table 32.5 PIO timings

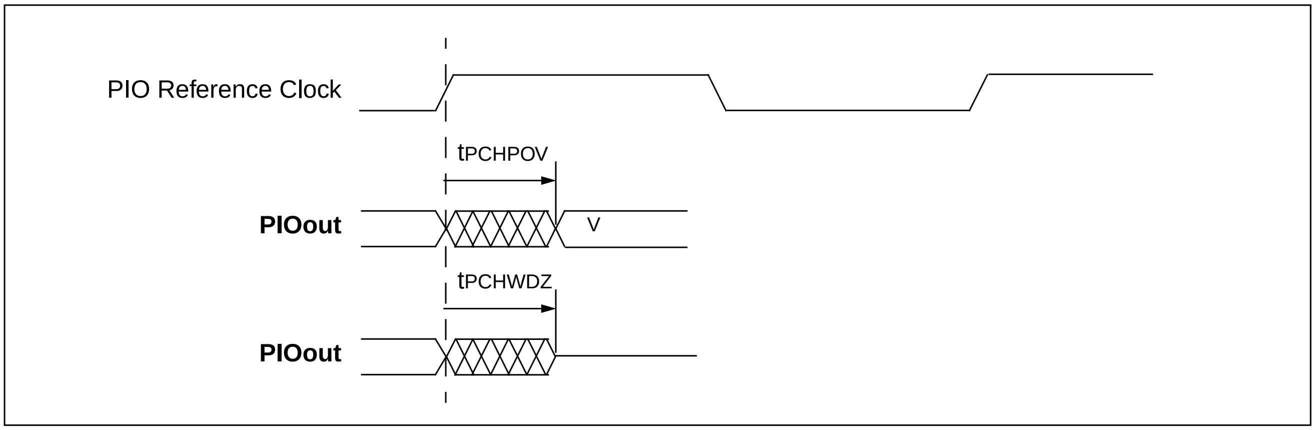


Figure 32.9 PIO timings

To be defined

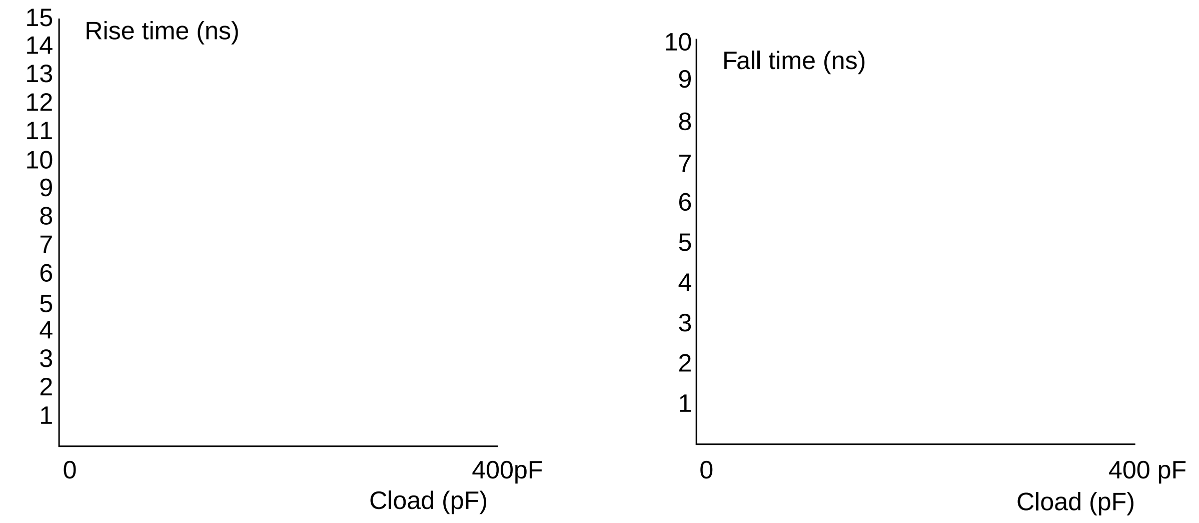


Figure 32.10 Rise and fall times

## 32.6 OS-Link timings

| Symbol | Parameter | Minimum | Nominal | Maximum | Units | Notes |
|---|---|---|---|---|---|---|
| $t_{JQR}$ | **LinkOut** rise time | | | 20 | ns | |
| $t_{JQF}$ | **LinkOut** fall time | | | 10 | ns | |
| $t_{JDR}$ | **LinkIn** rise time | | | 20 | ns | |
| $t_{JDF}$ | **LinkIn** fall time | | | 20 | ns | |
| $t_{JQJD}$ | Buffered edge delay | 0 | | | ns | |
| $\Delta t_{JB}$ | Variation in $t_{JQJD}$ 20 Mbits/s | | | 3 | ns | 1 |
| $C_{LIZ}$ | **LinkIn** capacitance @ f=1MHz | | | 10 | pF | |
| $C_{LL}$ | **LinkOut** load capacitance | | | 50 | pF | |

Table 32.6 OS-Link timings

### Notes

1 This is the variation in the total delay through buffers, transmission lines, differential receivers etc., caused by such things as short term variation in supply voltages and differences in delays for rising and falling edges.

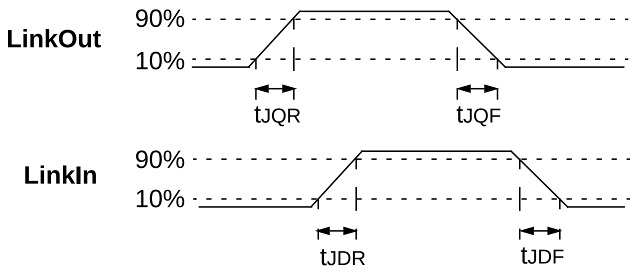


Figure 32.11 Link timings

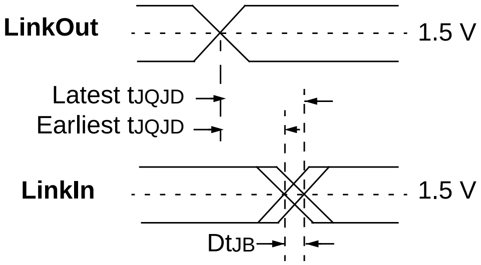


Figure 32.12 Buffered OS-Link timings

## 32.7 Reset and Analyse timings

| Symbol | Parameter | Minimum | Nominal | Maximum | Units |
|---|---|---|---|---|---|
| $t_{RSTLRSTH}$ | **not_Rst** pulse width low | 8 | | | ClockIn |
| $t_{RHRL}$ | **CPUReset** pulse width high | 1 | | | ClockIn |
| $t_{AHRH}$ | **CPUAnalyse** setup before **CPUReset** | 3 | | | ms |
| $t_{RLAL}$ | **CPUAnalyse** hold after **CPUReset** end | 1 | | | ClockIn |

Table 32.7 Reset and Analyse timings

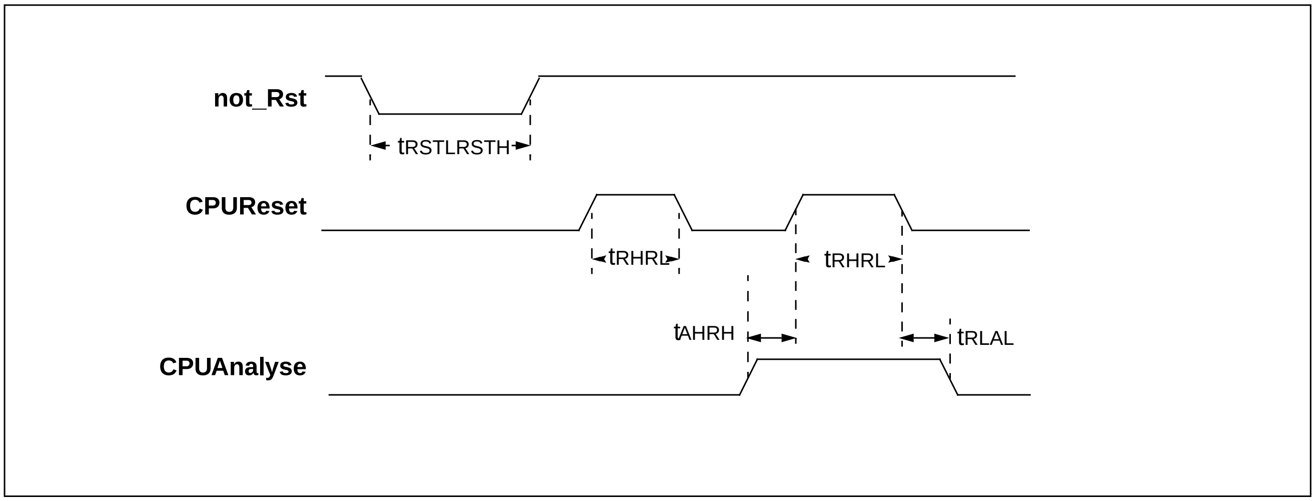


Figure 32.13 Reset and Analyse timings

## 32.8 Clock timings

| Symbol | Parameter | Minimum | Nominal | Maximum | Units | Notes |
|---|---|---|---|---|---|---|
| $t_{DCLDCH}$ | **ClockIn** pulse width low | 15.5 | 18.5 | 21.5 | ns | |
| $t_{DCHDCL}$ | **ClockIn** pulse width high | 15.5 | 18.5 | 21.5 | ns | |
| $t_{DCLDCL}$ | **ClockIn** period | | 37 | | ns | 1, 2 |
| $t_{DCR}$ | **ClockIn** rise time | | | | ns | 3 |
| $t_{DCF}$ | **ClockIn** fall time | | | | ns | 3 |

Table 32.8 ClockIn timings

### Notes

1 Measured between corresponding points on consecutive falling edges.

2 Variation of individual falling edges from their nominal times.

3 Clock transitions must be monotonic within the range $V_{IH}$ to $V_{IL}$.

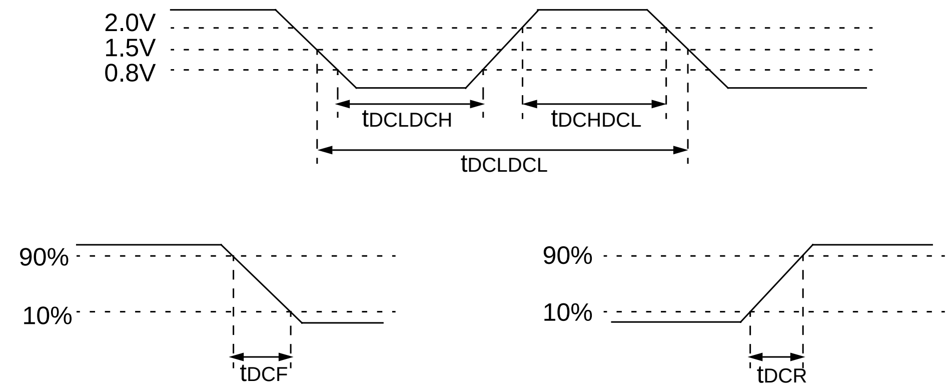


Figure 32.14 ClockIn timings

## 32.9 TAP timings

| Symbol | Parameter | Minimum | Nominal | Maximum | Units |
|---|---|---|---|---|---|
| $t_{TCHTCH}$ | TCK period | 50 | | | ns |
| $t_{TIVTCH}$ | TAP inputs valid to TCK high | 10 | | | ns |
| $t_{TCHTIX}$ | TAP input hold after TCK high | 10 | | | ns |
| $t_{TCHTOV}$ | TCK low to TAP output valid | | | 50 | ns |

Table 32.9  TAP timings

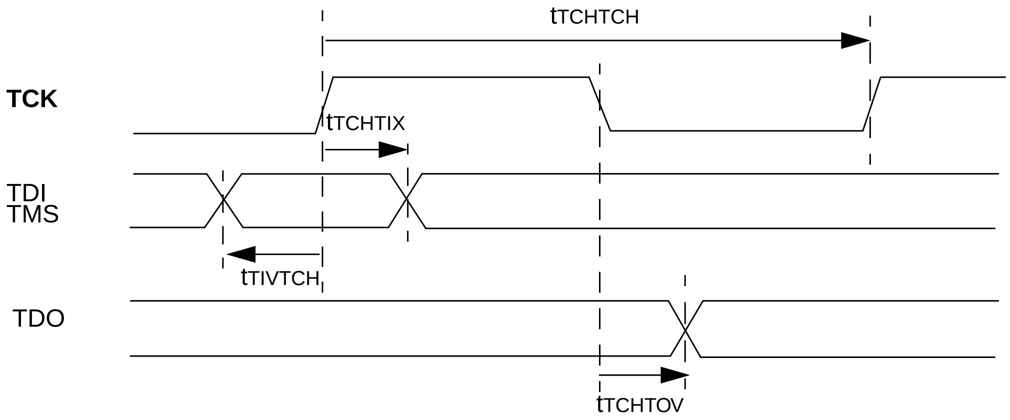


Figure 32.15 TAP timings

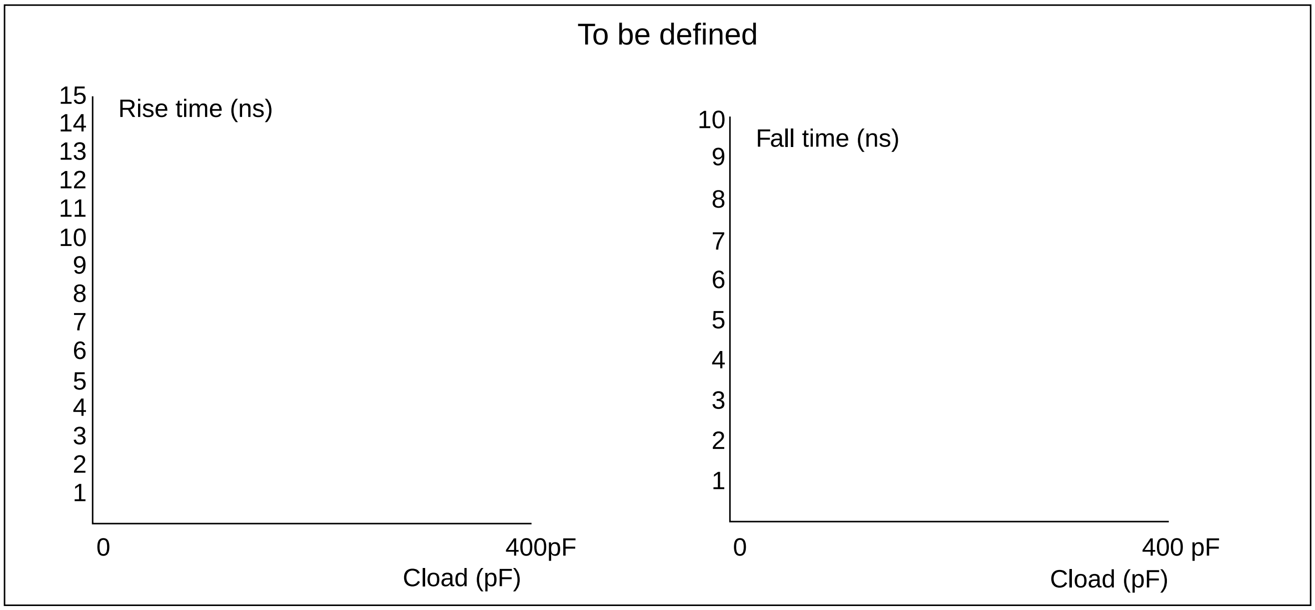


Figure 32.16 Rise and fall times

## 32.10 Transport stream demultiplexor timings

Transport stream demultiplexor timings for STi5500 silicon version D and later are given in Table 32.10. Timings for earlier versions are given in Table 32.11.

| Symbol | Parameter | Min | Nom | Max | Units | Notes |
|---|---|---|---|---|---|---|
| $t_{LCHLCH}$ | **F_B_CLK** period | 16 | | | ns | |
| $t_{LCHLCL}$ | **F_B_CLK** pulse width high | 2 | | | ns | |
| $t_{LCLLCH}$ | **F_B_CLK** pulse width low | 2 | | | ns | |
| $t_{LDVLCH}$ | Transport stream demultiplexor signals valid to **F_B_CLK** low | 2.5 | | | ns | |
| $t_{LCHLDX}$ | Transport stream demultiplexor signals hold after **F_B_CLK** low | 2 | | | ns | |

Table 32.10 Transport stream demultiplexor timing for STi5500 silicon version D and later

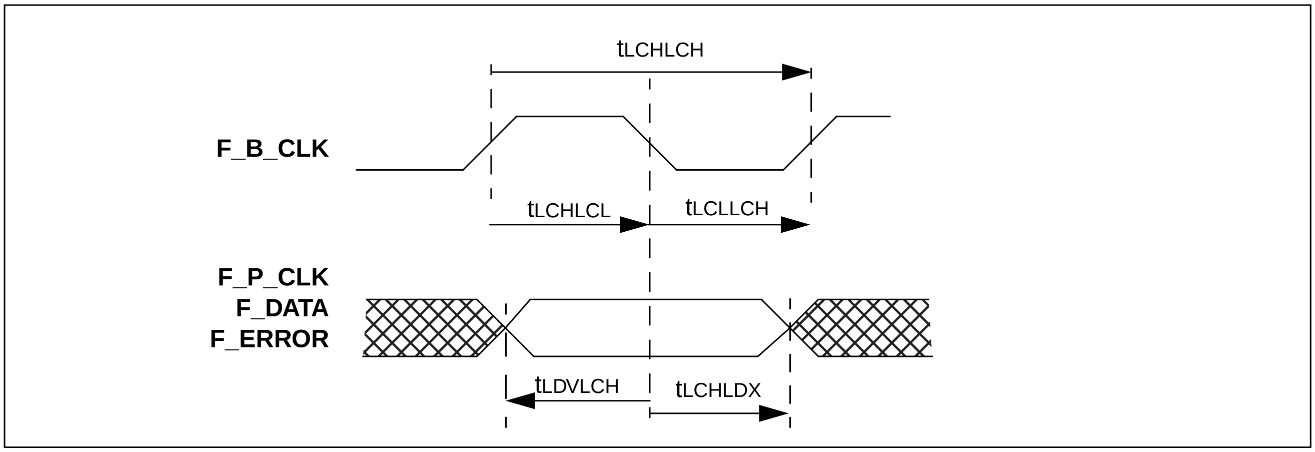


Figure 32.17 Transport stream demultiplexor timing

| Symbol | Parameter | Min | Nom | Max | Units | Notes |
|---|---|---|---|---|---|---|
| tLCHLCH | **F_B_CLK** period | 16 | | | ns | |
| tLCHLCL | **F_B_CLK** pulse width high | 2 | | | ns | |
| tLCLLCH | **F_B_CLK** pulse width low | 2 | | | ns | |
| tLDVLCH | **F_DATA** signal valid to **F_B_CLK** low | 2.5 | | | ns | |
| tLDVLCH | **F_P_CLK** signal valid to **F_B_CLK** low | 2.5 | | | ns | |
| tLDVLCH | **F_ERR** signal valid to **F_B_CLK** low | 5.0 | | | ns | |
| tLCHLDX | **F_DATA** signal hold after **F_B_CLK** low | 3.5 | | | ns | |
| tLCHLDX | **F_P_CLK** signal hold after **F_B_CLK** low | 3.0 | | | ns | |
| tLCHLDX | **F_ERR** signal hold after **F_B_CLK** low | 2 | | | ns | |

Table 32.11  Transport stream demultiplexor timing for STi5500 silicon versions before D

# 33 Electrical specifications

The STi5500 is being characterized. These electrical specifications are given for guidance only and will be updated when characterization is complete.

## 33.1 Absolute maximum ratings

Stresses greater than those listed in Table 33.1 may cause permanent damage to the device. This is a stress rating only and functional operation of the device at these or any other conditions above those indicated in the operating sections of this specification is not implied. Continuous operation at these limits is not intended and should be limited to those conditions specified in section 33.2.

| Symbol | Parameter | Min | Max | Unit |
|---|---|---|---|---|
| $V_{DD}$, $V_{DDA}$ | DC supply voltage | | 4.5 | V |
| $V_I$ | Voltage on input, bi-directional and address pins | GND-0.6 | 6.5 | V |
| $V_O$ | Voltage on output pins | GND-0.6 | VDD+0.6 | V |
| $I_O$ | DC output current | | 25 | mA |
| $T_S$ | Storage temperature (ambient) | -55 | 125 | °C |
| $T_A$ | Temperature under bias (ambient) | -55 | 125 | °C |

Table 33.1 Absolute maximum ratings

## 33.2 Operating conditions

The following specifications are over $V_{DD}$ = 3.3V ± 0.3V, $T_A$ = 0˚C to 70˚C unless otherwise specified.

| Symbol | Parameter | Min | Max | Units | Notes |
|---|---|---|---|---|---|
| $V_I$, $V_O$ | Input or output voltage | 0 | 5.75 | V | |
| $C_L$ | Load capacitance per pin | | 60 | pF | |
| $C_{LD}$ | Load capacitance per data pin | | 60 | pF | |
| $C_{LA}$ | Load capacitance per address/strobe pin | | 100 | pF | |
| $C_{LP}$ | Load capacitance per PIO pin | | 400 | pF | |
| $T_A$ | Operating temperature (ambient) | 0 | 70 | °C | |
| PD | Power dissipation | | 2 | W | Measured at 40 MHz with no static loads on the EMI pins and with a 30 pF load on all output pins. |

Table 33.2 Operating conditions

## 33.3 DC electrical characteristics

| Symbol | Parameter | Min | Typ | Max | Units | Notes |
|---|---|---|---|---|---|---|
| VDD | Positive supply voltage | 3.0 | 3.3 | 3.6 | V | |
| VIH | Input logic 1 voltage | 2.0 | | 5.5 | V | |
| VIL | Input logic 0 voltage | -0.5 | | 0.8 | V | |
| IIN | Input current (input pin) | | | ±10 | µA | 0 ≤ VI ≤ 5.5 |
| IOZ | Off state digital output current | | | ±50 | µA | 0 ≤ VI ≤ VDD and 4.5 < VI < 5.5 |
| IOZP | Peak off state output current | | | ±200 | µA | VDD ≤ VI ≤ 4.5 |
| VOH | Output logic 1 voltage | 2.4 | | | V | At rated output load currents |
| VOL | Output logic 0 voltage | | | 0.4 | V | At rated output load currents |
| CIN | Input capacitance (input pins) | | | 10 | pF | |
| CIO | Input capacitance (bi-directional pins) | | | 15 | pF | |
| COUT | Output capacitance | | | 15 | pF | |

Table 33.3 DC specifications

| Symbol | Parameter | Min | Typ | Max | Units | Notes |
|---|---|---|---|---|---|---|
| VDDA | Analog positive supply voltage | 3.0 | 3.3 | 3.6 | V | |
| IDDA | Analog current consumption | 20 | | 50 | mA | |
| RIREF | Resistance for reference current source for 3 D/A converters | | 1.2 | | kΩ | |
| VO | Output voltage Dyn | 0.95 | | 1.10 | VPP | |
| | DAC to DAC VO max code | | | 3 | % | |
| ILE | LF integral non-linearity | | ± 1 | | LSB | |
| DLE | LF differential non-linearity | | ± 0.5 | | LSB | |

Table 33.4 DC specifications

The electrical types of the pins are given in Table 33.5. The table gives the 5V tolerance and type for each physical pin, listed in alphabetical order of pin name. The meanings of the type codes are given in Table 33.6.

| Pin name | 5V tol’nce | Pin electrical type | |
|---|---|---|---|
| | | Input | Output |
| **A_C_Req** | yes | S | |
| **A_Pts_Stb** | yes | S | |
| **Ad0-11** | no | T8 | |
| **Adr1-21** | yes | T6 | |
| **B_Out** | no | A | |
| **Brm0 / Oslink_Sel** | yes | S | T6 |
| **Brm1 / BooFromRom** | yes | S | T6 |
| **Brm2** | no | T8 | |
| **C_Out** | no | A | |
| **ProcClockOut** | yes | T6 | |
| **CV_Out** | no | A | |
| **Data0-15** | yes | S | T6 |
| **Dq0-15** | no | S | T8 |
| **Dqml** | no | T8 | |
| **Dqmu** | no | T8 | |
| **F_B_Clk** | yes | S | |
| **F_Data** | yes | S | |
| **F_Error / P_Start** | yes | S | |
| **F_P_Clk / D_Valid** | yes | S | |
| **G_Out** | no | A | |
| **Gnd** | | P | |
| **I_Ref_Dac_RGB** | no | A | |
| **I_Ref_Dac_YCC** | no | A | |
| **Irq0-2** | yes | S | |
| **Link_Ext_Clk** | yes | S | |
| **lrclk / A_Word_Clk** | no | C8 | |
| **MemClkIn** | no | S | |
| **MemClkOut** | no | T8 | |
| **MemWait** | no | S | |
| **not_Cas0-1** | yes | T6 | |
| **not_CE1-3** | yes | T6 | |
| **not_Hsync** | yes | S | T6 |

Table 33.5  Pin electrical type

| Pin name | 5V tol'nce | Pin electrical type | |
|---|---|---|---|
| | | Input | Output |
| not_OE | yes | T6 | |
| not_Ras0 / not_CE0 | yes | T6 | |
| not_Ras1 | yes | T6 | |
| not_Rst | yes | S | |
| not_SdCas | no | T8 | |
| not_SdCS0-1 | no | T8 | |
| not_SdRas | no | T8 | |
| not_SdWE | no | T8 | |
| not_Trst | yes | S | |
| not_WE0-1 | yes | T6 | |
| Nrss_Clk | no | T6 | |
| Nrss_In | yes | S | |
| Nrss_Out | no | T8 | |
| Odd_not_Even | yes | S | T6 |
| Osc_In / 27Mhz_Out | yes | S | T6 |
| Osd_Active | yes | S | T6 |
| Pcm_ClkIn | yes | S | T6 |
| Pcm_ClkOut / A_C_Stb | no | C8 | |
| Pcm_Data / A_C_Data | no | C8 | |
| Pio0_0-7 | yes | S | T6 |
| Pio1_0-7 | yes | S | T6 |
| Pio2_0-7 | yes | S | T6 |
| Pio3_0-7 | yes | S | T6 |
| Pio4_0-7 | yes | S | T6 |
| PixClk_27Mhz | yes | S | |
| R_Out | no | A | |
| ReadnotWrite | yes | T6 | |
| Sdav_Clk | yes | S | T6 |
| Sdav_Data | yes | S | T6 |
| Sdav_Dir | yes | S | T6 |
| Tck | yes | S | |
| Tdi | yes | S | |

Table 33.5 Pin electrical type

| **Pin name** | **5V tol'nce** | **Pin electrical type** | |
|---|---|---|---|
| | | **Input** | **Output** |
| **Tdo** | yes | T6 | |
| **Tms** | yes | S | |
| **V_Ref_Dac_RGB** | no | A | |
| **V_Ref_Dac_YCC** | no | A | |
| **Vdd** | - | P | |
| **Vdda_0-1** | - | P | |
| **Vssa_0-1** | - | P | |
| **Y_Out** | no | A | |

Table 33.5 Pin electrical type

| **Pin type code** | **Pin type** |
|---|---|
| P | Power |
| A | Analog |
| T6 | TTL, 6mA |
| T8 | TTL, 8mA |
| C8 | CMOS, 8mA |
| S | Schmitt trigger |

Table 33.6 Pin type codes in Table 33.5

# Appendix A Channel model

The STi5500 on-chip bus which connects the ST20 processor core and the other modules provides a unique way of communicating between data processing/interface modules, the CPU and memory (both on and off chip).

The model relies on three main elements of the system. The microkernel of the CPU, the interconnect protocol, and the design of the module. Instructions are provided which enable the programmer to make use of these features in a simple and flexible way.

The CPU uses a group of reserved locations at the base of memory to store the task identifier of a task using one of the channels, see the memory map for details. When a task performs an instruction requiring communication via the channel the task identifier is stored in the channel location (specified by the instruction operand) and the appropriate command (determined by the instruction) is sent to the module. This task is now considered inactive and will take no further CPU time. The microkernel will begin executing the next active task from its queue. When the module has completed the command, an acknowledge is sent to the CPU which signals the microkernel to remove the task identifier from the channel location and put it on the back of the queue of active processes waiting for CPU time.

The type of operations this is used for is data transfers into and out of CPU memory. This method of communication has the advantage that the speed and overhead of the data transfer are not taking up CPU time. The close coupling of the microkernel and these protocols means that the set-up, acknowledge and context switch times are very short, less than 500 ns in most cases.

## A.1 Example

The CPU executes an *in* instruction from the Block Move DMA module. Operands to the *in* instruction are the base pointer in CPU memory and the size in bytes. The task ID of the task executing the *in* instruction is placed in address #80000034. The internal bus sends the channel number, the *in* command, the base pointer and the size. This will be received by the correct module using the channel number. The CPU is now free to continue with another operation. The Block Move DMA module will now input ‘size’ bytes of data and place them in the addresses above the base pointer. When the correct number of bytes have been received the module returns an acknowledge command and the channel number to the CPU. The microkernel takes the task ID from address #80000034 and adds it to the back of the active list.

Information furnished is believed to be accurate and reliable. However, STMicroelectronics assumes no responsibility for the consequences of use of such information nor for any infringement of patents or other rights of third parties which may result from its use. No license is granted by implication or otherwise under any patent or patent rights of STMicroelectronics. Specifications mentioned in this publication are subject to change without notice. This publication supersedes and replaces all information previously supplied. STMicroelectronics products are not authorized for use as critical components in life support devices or systems without express written approval of STMicroelectronics.

STMicroelectronics GROUP OF COMPANIES
Australia - Brazil - Canada - China - France - Germany - Italy - Japan - Korea - Malaysia - Malta - Mexico - Morocco
The Netherlands - Singapore - Spain - Sweden - Switzerland - Taiwan - Thailand - United Kingdom - U.S.A.

**http://www.st.com**