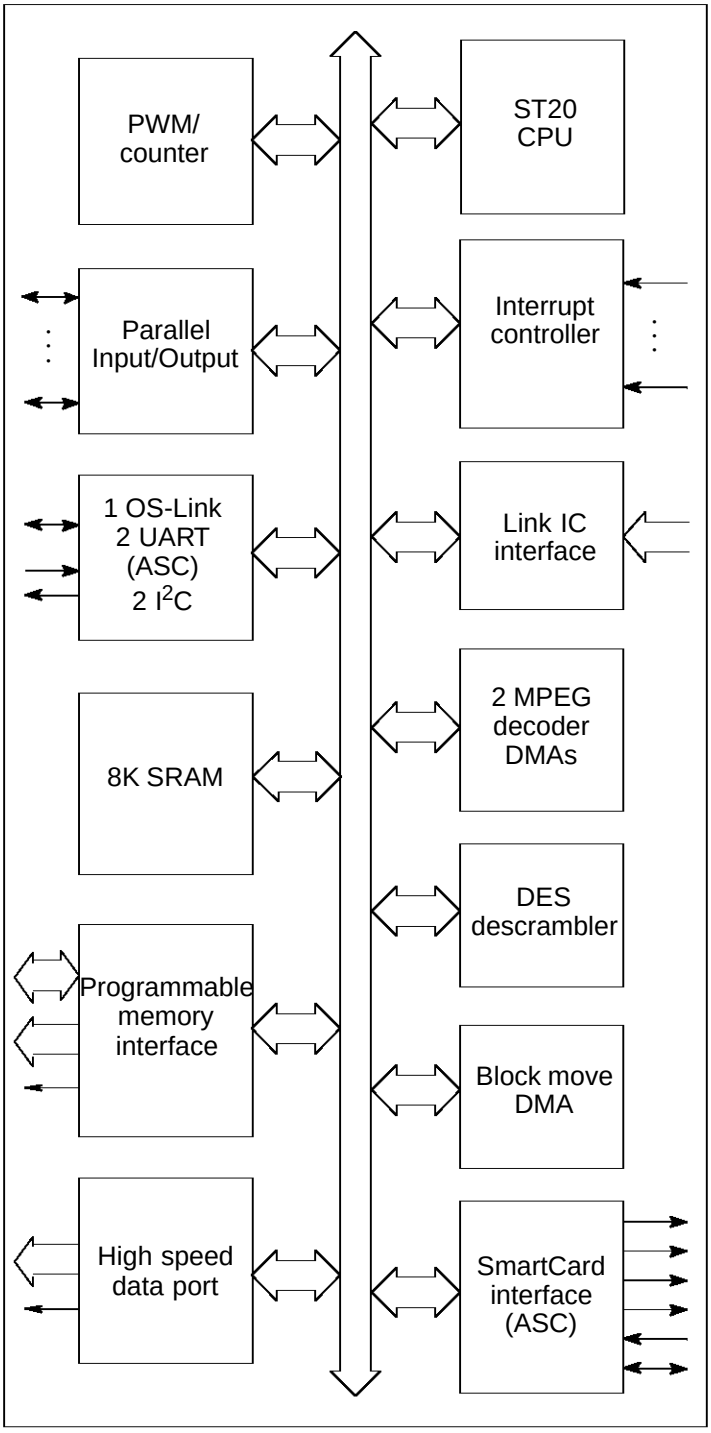
SGS-THOMSON MICROELECTRONICS

ST20-TP1

PROGRAMMABLE TRANSPORT IC FOR DSS APPLICATIONS

FEATURES

- Enhanced 32-bit VL-RISC CPU
  - 0 to 40 MHz processor clock
  - fast integer/bit operations
  - very high code density
- 8 Kbytes on-chip SRAM
  - 160 Mbytes/s maximum bandwidth
- Programmable memory interface
  - 4 separately configurable regions
  - 8/16/32 bits wide
  - support for mixed memory
  - 2 cycle external access
  - support for page mode DRAM
  - support for MPEG decoders
- Serial communications
  - OS-Link
  - 2 Programmable UARTs (ASC)
  - 2 Synchronous serial interfaces (I^2C)
- Vectored interrupt subsystem
  - Prioritized interrupts
  - 8 levels of preemption
  - 500 ns response time
- DMA engines/interfaces
  - Link IC DMA interface
  - DES descrambler DMA
  - Block move DMA
  - 2 MPEG decoder DMAs
  - High speed data port DMA
  - SmartCard interface
- PWM/counter module
  - Two 8-bit PWM
  - Two 32-bit counters and capture registers
- Programmable IO module
  - 32 fully programmable IO bits
- Professional toolset support
  - ANSI C compiler and libraries
  - INQUEST advanced debugging tools
- Technology
  - 0.5 micron process technology
- 208 pin PQFP package
- JTAG Test Access Port



APPLICATIONS

- Set top terminals

Contents

1 Introduction

The ST20-TP1 is a programmable transport IC for digital set top boxes. It has been designed to meet the requirements of the specification for DSS™ Digital Satellite Systems[1].

The ST20-TP1 combines the functionality of the set top box transport IC and system microcontroller in to a single device. The performance offered by the ST20 32-bit micro-core allows the following operations to be performed concurrently in software:

1 Transport layer demultiplexing

2 Service data filtering

3 Device drivers and clock synchronization

4 Electronic program guide

5 System management functions

6 Conditional access

Note: Source code software licences are available from SGS-THOMSON for modules 1, 2 and 3 above.

The advantages of using software versus dedicated hardware for these functions are two-fold:

- Flexibility — it is quick and simple to modify software to adapt to a new system requirement or to a change in a standard.
- Upgradability — the use of a 32-bit CPU enables the use of advanced graphics routines for on-screen display functions and enables fast turnaround system upgrades.

The ST20 micro-core family has been developed by SGS-THOMSON Microelectronics 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 ST20-TP1 uses the ST20 macrocell library to provide all of the dedicated hardware modules required in a DSS set top box programmable transport-IC. These include:

- Link-IC interface to MPEG transport stream
- I^2C 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
- 8 Kbytes of internal SRAM
- DMA module to MPEG audio and video device(s)
- External memory interface supporting DRAM, EPROM and peripherals
- PWM/timer module for control of VCXO and system clocking

1. DSS and Digital Satellite System are trademarks of Hughes Communications a unit of GM Hughes Electronics.

- Programmable I/O pins
- DES descrambler
- Smart card interface

The ST20-TP1 has also been designed to minimize system costs. The memory interface module contains a zero glue logic DRAM controller, a low cost 8-bit EPROM interface and a port for connecting directly to the MPEG audio and video devices. Furthermore the ST20 VL-RISC micro-core has the highest code density of any 32-bit CPU, leading to the lowest cost program ROM.

The ST20-TP1 is supported by a range of software and hardware development tools for PC and UNIX hosts including an ANSI-C ST20 software toolset incorporating the ST20 INQUEST window based debugger.

2 ST20-TP1 architecture overview

A block diagram of a typical digital set top receiver using the ST20-TP1 is shown in Figure 2.1.

The ST20-TP1 performs the system microcontroller and transport demultiplexer functions. It has been designed to directly interface to external memory and peripherals with no extra glue logic, keeping the system cost to a minimum. The ST20-TP1 architectural block diagram is shown in Figure 2.2.

2.1 Transport demultiplexing

The transport demultiplexing function is performed in a mixture of hardware and software. Typical operation is as described below.

Data packets from the Link-IC are input into memory by the Link-IC interface using DMA. The packet is parsed in software to determine its type and to extract data from it. If the packet is encrypted using the Data Encryption Standard (DES), a memory to memory DMA operation through the DES decryption controller (DESC) is performed before the packet can be parsed.

After parsing the packet, the data is either transferred to buffers in external memory or passed to other software tasks as a message. The transfer from internal to external memory can also be performed as a memory to memory DMA operation using the block move mode of the DESC.

Audio or Video MPEG compressed data extracted from the input data packets is transferred to the decoders using two independent DMA controllers. These read data from memory and then write it to a decoder in response to a DMA request from the decoder.

The unique architecture of the ST20 family, in particular the scheduler implemented in microcode, allows the transport demultiplex functions to typically occupy less than half the available CPU cycles.

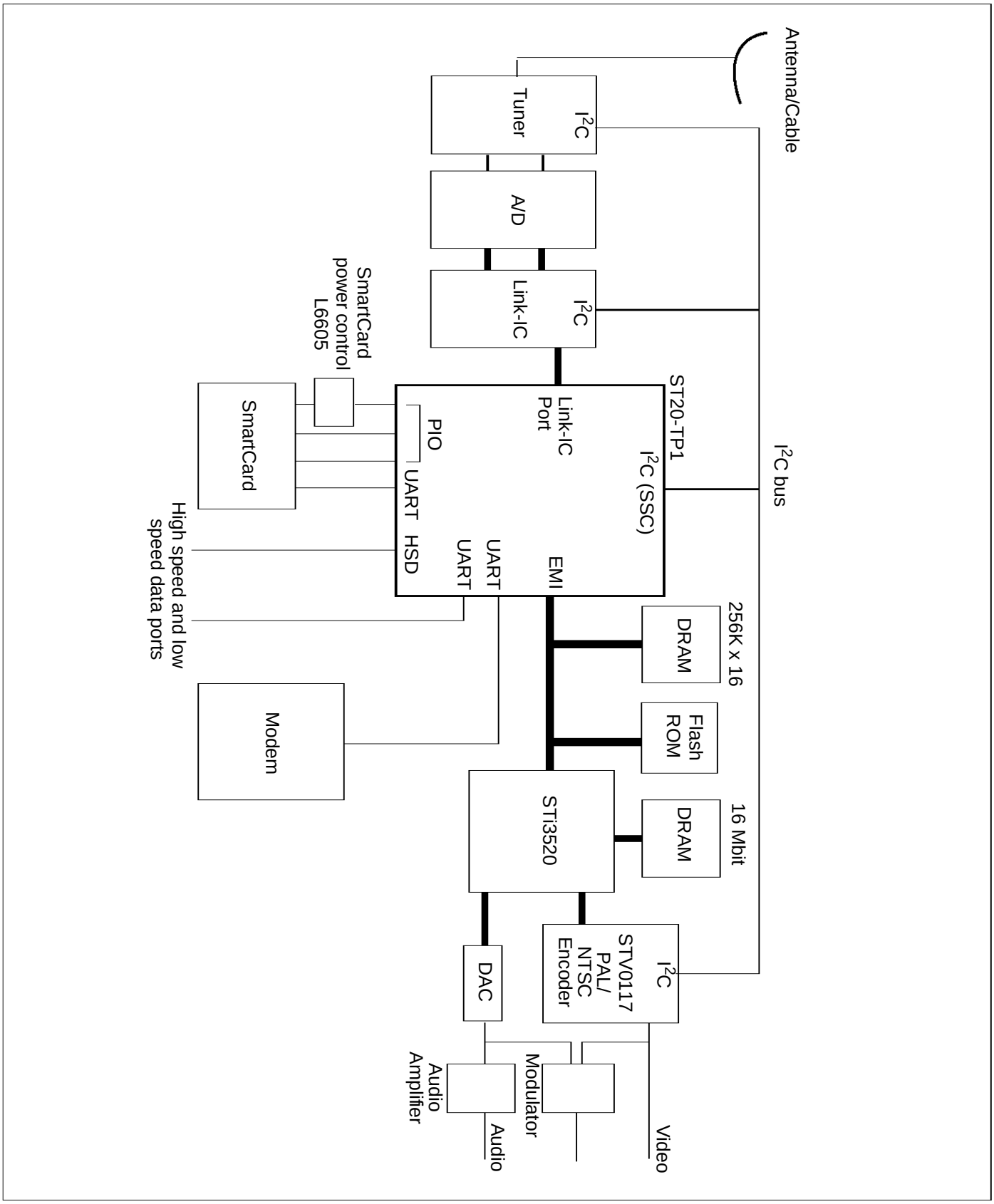


Figure 2.1 Digital set top box block diagram

2.2 ST20-TP1 functional modules

Figure 2.2 shows the subsystem modules that comprise the ST20-TP1. These modules are outlined below and more detailed information is given in the following chapters of this datasheet.

CPU

The Central Processing Unit (CPU) on the ST20-TP1 is the ST20 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 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).

Memory subsystem

The ST20-TP1 on-chip memory system provides 160 Mbytes/s internal data bandwidth, supporting pipelined 2-cycle internal memory access at 25 ns cycle times. The ST20-TP1 memory system consists of SRAM and an external memory interface (EMI).

The ST20-TP1 product has 8 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. Furthermore small systems could place all code and data on-chip, increasing performance and reducing system cost. For the transport layer demultiplexing functions calculations have shown that the code can fit in internal memory together with its stack and packet buffers. This gives the required performance for these functions.

The ST20-TP1 EMI controls access to the external memory and peripherals including the MPEG decoder registers and DMA data ports. Special strobes have been added to one of the banks of the EMI to allow a direct interface to the SGS-THOMSON microelectronics range of MPEG2 audio and video decoders.

The ST20-TP1 EMI can access a 16 Mbyte (or greater if DRAM is used) physical address space in each of the three general purpose memory banks, and provides sustained transfer rates of up to 80 Mbytes/s for SRAM, and up to 40 Mbytes/s using page-mode DRAM. The EMI includes programmable strobes to support direct interfacing to MPEG decoder devices.

System services module

The ST20-TP1 system services module includes:

- reset, initialization and error port.
- 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.

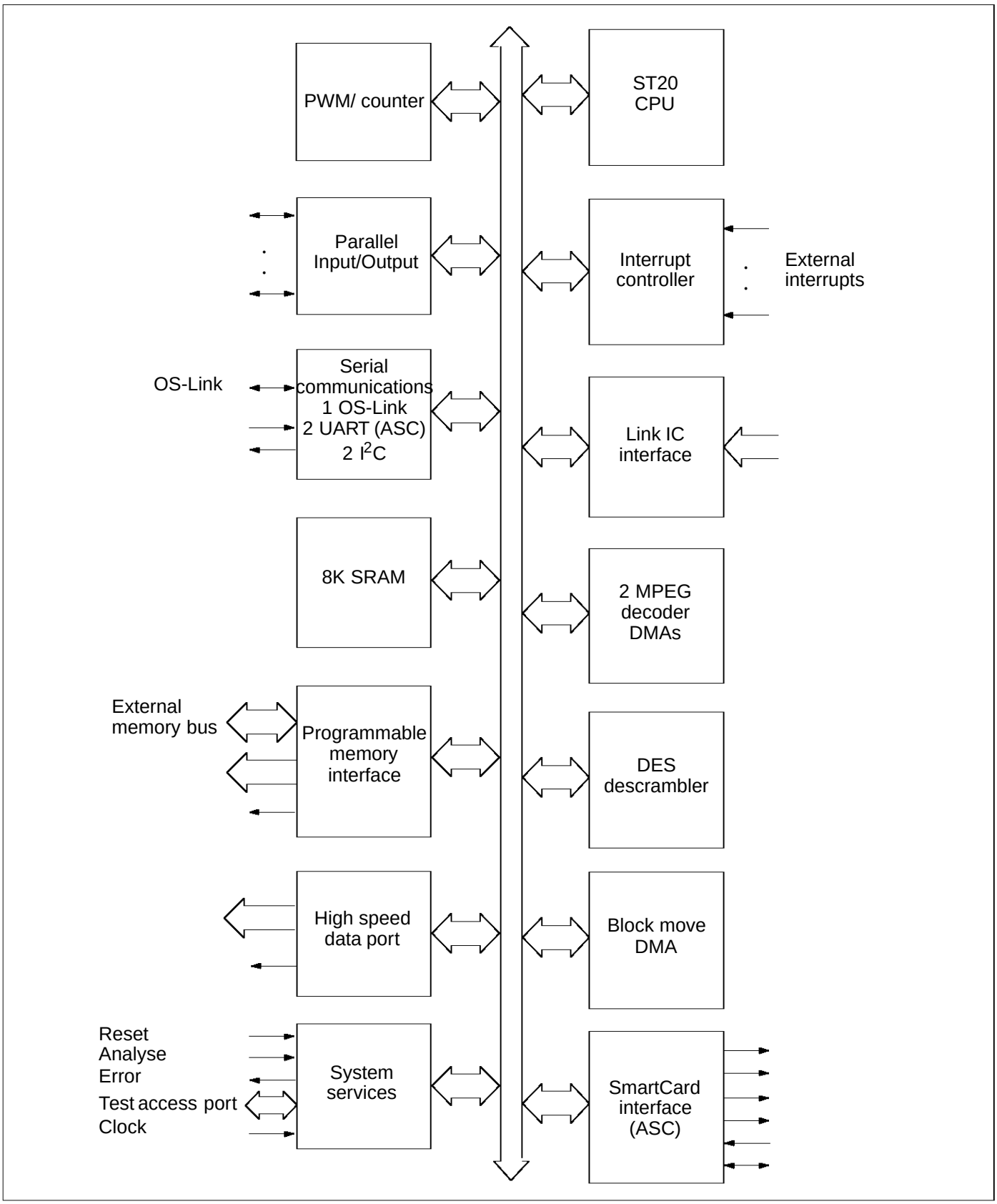


Figure 2.2 ST20-TP1 architectural block diagram

Serial communications

To facilitate the connection of this system to a modem for a pay-per-view type system and other peripherals, two UARTs (Asynchronous Serial Controllers (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.

Two high-speed Synchronous Serial Controllers (SSC) are provided on the device. These can be used to interface to a wide variety of serial memories, remote control receivers, and other micro-controllers. Various interface standards exist for these, the most important of which is the I^2C bus in the set-top box application as this is the interface used most often for the control of the Link-IC and the PAL/NTSC encoder.

The ST20-TP1 has an OS-Link based serial communications subsystem. OS-Links use an asynchronous bit-serial (byte-stream) protocol, each bit received is sampled five times, hence the term *oversampled links* (OS-Links). An OS-Link provides a pair of channels, one input and one output channel.

There is one OS-Link on the ST20-TP1 which acts as a DMA engine independent of the CPU. The link is used for:

- bootstrapping during development
- debugging

Interrupt subsystem

The ST20-TP1 interrupt subsystem supports eight prioritized interrupt levels. In addition, there is an interrupt level controller which multiplexes fourteen incoming interrupts onto the eight programmable interrupt levels. This multiplexing is controllable by software.

Four 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.

Link IC interface

The Link-IC interface provides a byte wide data input from the Link-IC. The interface between the CPU and this module is provided using a channel interface allowing data transfer from the link IC to memory independently of the CPU. Using a channel interface requires a low CPU overhead at the start and end of each transfer.

DES decryption and block move

The Data Encryption Standard (DES) decryption is supported by the DES description controller (DESC) module. This can be used to decrypt and move blocks of data from one area of memory to another using DMA operations. The module implements DES decryption in Electronic Code Book (ECB) mode.

Block move engine

The transfer of data between two areas of memory can also be performed as a memory to memory DMA operation using the block move module.

MPEG DMAs

The two MPEG DMA controllers are used to transfer MPEG compressed data from the memory to the decoder chips. DMA strobes are provided by the EMI to support the direct connection of decoder ICs to the ST20-TP1.

High speed data port

A byte wide parallel interface supports a high speed data output port from the set top receiver. The interface has a dedicated DMA controller to transfer data from memory to the port with little CPU overhead.

SmartCard interface

The SmartCard interface supports SmartCards that are compliant with ISO7816-3 and use the asynchronous protocol. The interface is implemented with a third UART (ASC), dedicated programmable clock generator, and eight bits of parallel IO port.

PWM and Counter module

This unit includes two separate pulse width modulator (PWM) generators and two counters with capture registers. The counters can be clocked from a pre-scaled internal clock or from a pre-scaled external clock via the **CaptureClock** 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 counters are 32 bits with 32-bit capture registers.

Parallel IO module

Thirty-two 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 ST20-TP1 device are multi-function and can either be configured as PIO or connected to an internal peripheral signal.

3 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 preemption. In addition, multi-level preemption is provided by the interrupt subsystem, see Chapter 4 for details. Additionally, there is a per-priority trap handler to improve the support for arithmetic errors and illegal instructions, refer to section 3.6.

3.1 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 (**IptrReg**) which points to the next instruction to be executed.
- The status register (**StatusReg**).
- 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.

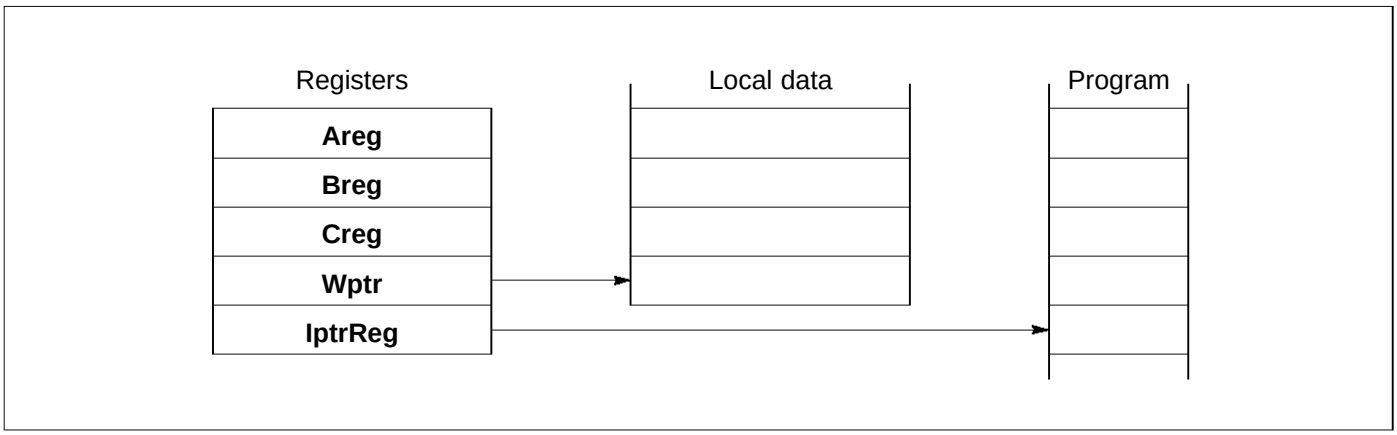


Figure 3.1 Registers used in sequential integer processes

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.

3.2 Processes and concurrency

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, installing a user scheduler, etc.

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 por-

tion of the processor's time to each active low priority process in turn (see Section 4.3). Active processes waiting to be executed are held in two linked lists of process workspaces, 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 4.2, 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.

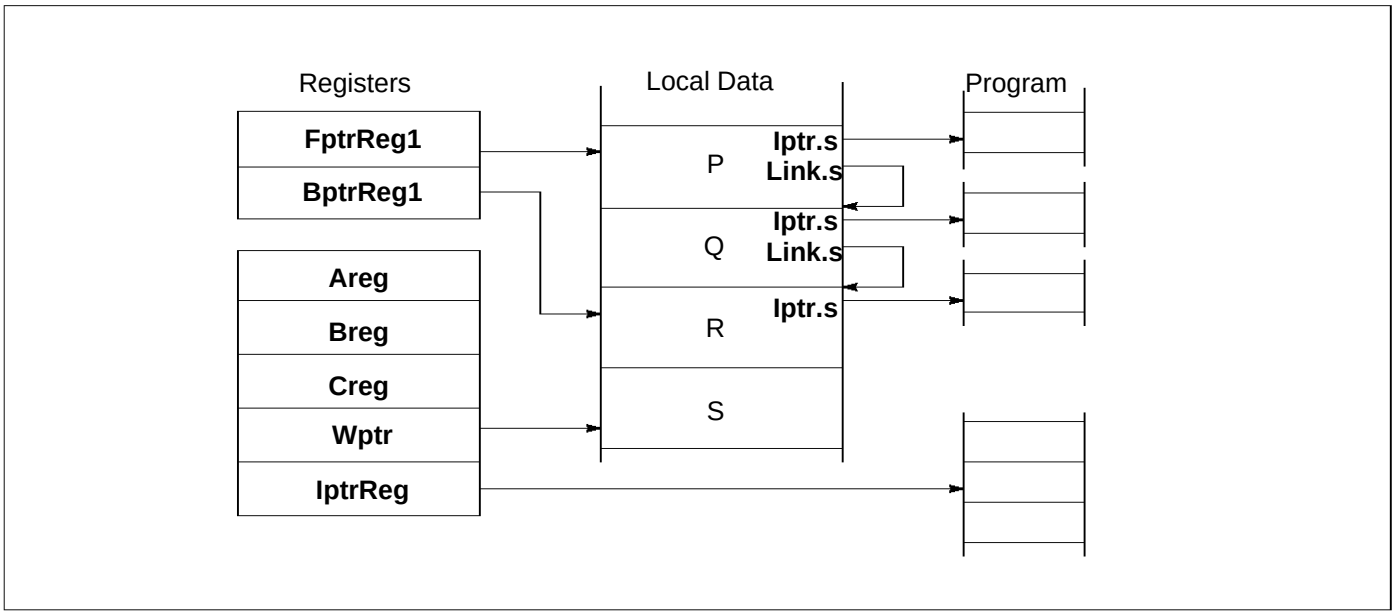


Figure 3.2 Linked process list

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

Table 3.1 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 con-

struct, *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 preempt 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.

3.3 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 computationally intensive tasks.

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 CPU's time; 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 whilst 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 reloaded 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 Table 6.21 on page 44). These instructions allow extensions to be made to the scheduler for custom runtime 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 are abortable and are restarted when the process next becomes active (refer to the Instruction Set chapter).

3.4 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.

3.5 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 is incremented approximately every 64 microseconds, giving 15625 ticks in one second. It has a full period of approximately 76 hours. Timer frequencies are approximate and depend on the processor speed selection (see section 12.1).

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 3.2 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 4.3 shows two processes waiting on the timer queue, one waiting for time 21, the other for time 31.

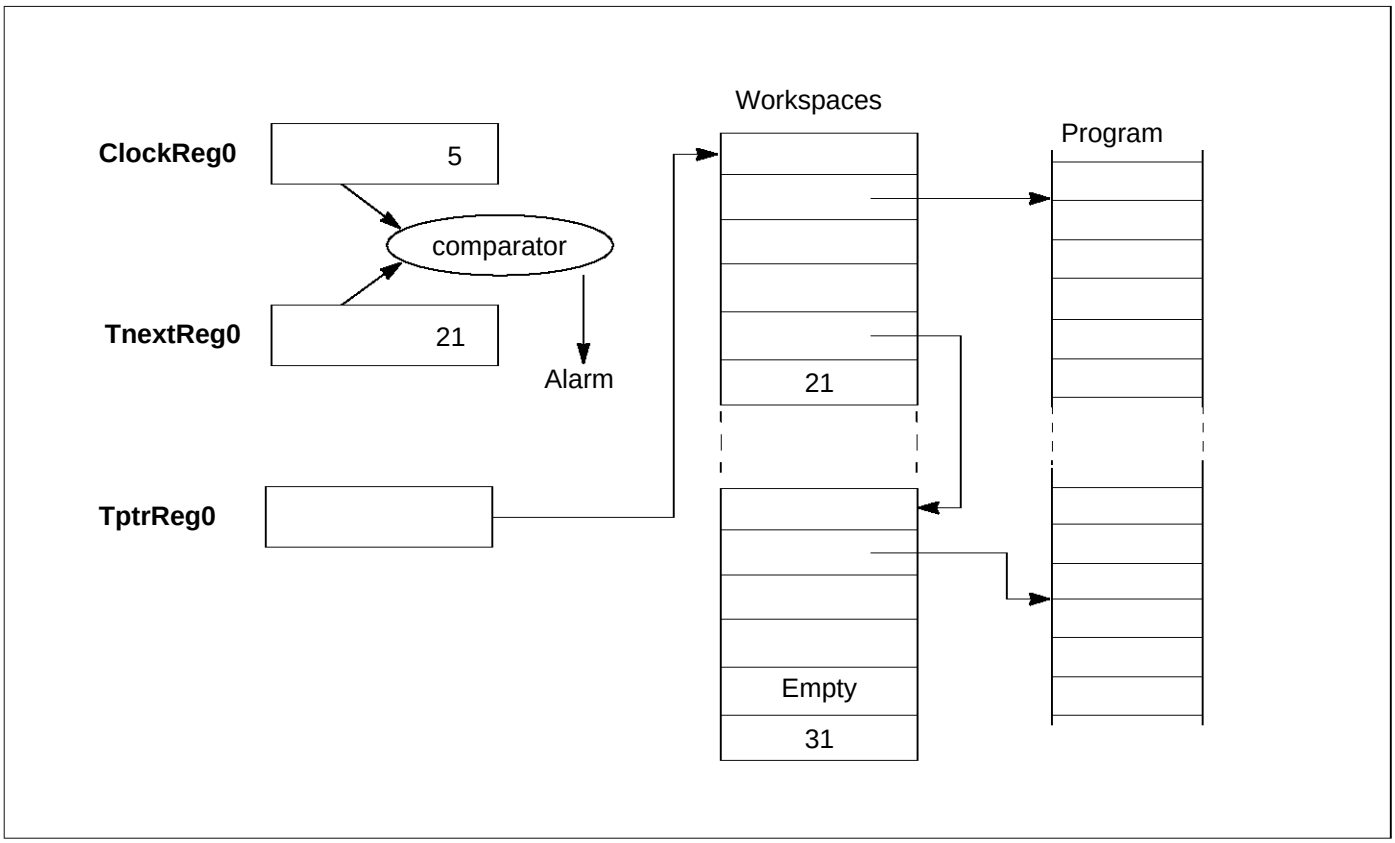


Figure 3.3 Timer registers

3.6 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 are individually maskable.

3.6.1 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 4.4.

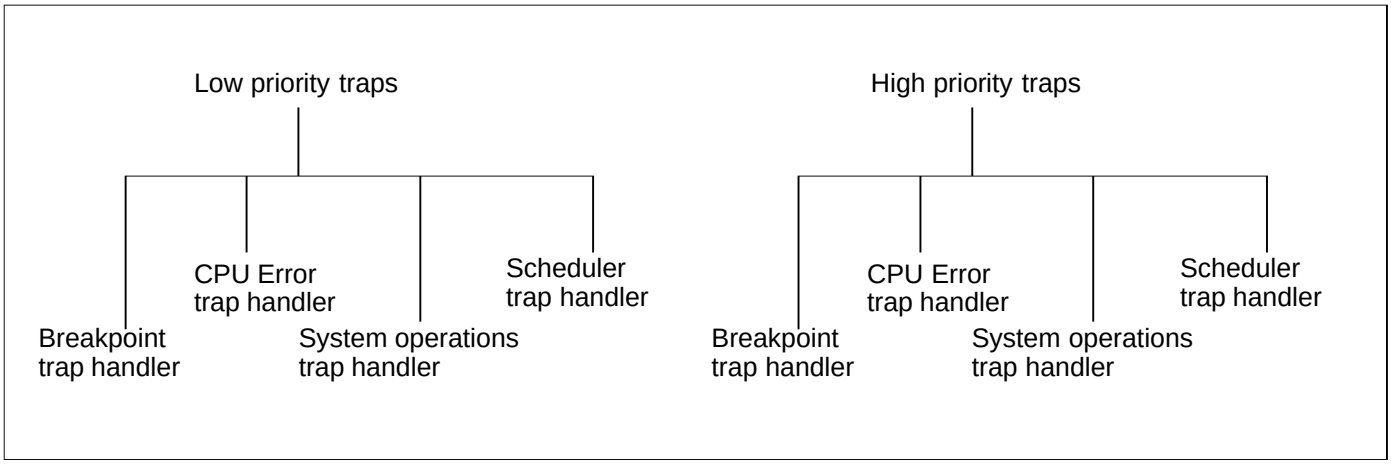


Figure 3.4 Trap arrangement

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 micro-scheduler rather than by an executing process.

Note, when the scheduler trap is caused by a process that is ready to be scheduled, the **Wptr** of that process is stored in the workspace of the scheduler trap handler, at address 0. The trap handler can access this using a *ldl 0* instruction.

Trap groups encoding is shown in Table 4.4 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 3.3 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.

3.6.2 Events that can cause traps

Table 3.4 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* (operate) is executed with an invalid operand.
LoadTrap	4	2	When the trap descriptor is read with the *ldtraph* (load trap handler) instruction or when the trapped process status is read with the *ldtrapped* (load trapped) instruction.
StoreTrap	5	2	When the trap descriptor is written with the *sttraph* (store trap handler) instruction or when the trapped process status is written with the *sttrapped* (store trapped) 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 3.4 Trap causes and **Status**/**Enable** codes

3.6.3 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 3.6.4.

The trap handler structure specifies what should happen when a trap condition is present, see Table 3.5.

	Comments	
Iptr	**Iptr** of trap handler process.	Base + 3
Wptr	**Wptr** of trap handler process.	Base + 2
Status	Contains the **Status** register that the trap handler starts with.	Base + 1
Enables	Contains 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 3.5 Trap handler structure

The trapped process structure saves some of the state of the process that was running when the trap was taken, see Table 3.6.

	Comments	
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 4.5 for trap codes.	Base + 1
Enables	Interrupt enables.	Base + 0

Table 3.6 Trapped process structure

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 3.4.

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 3.6.4). 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).

3.6.4 Trap instructions

Trap handlers and trapped processes can be set up and examined via the *ldtraph*, *sttraph*, *ldtrapped* and *sttrapped* instructions. Table 3.7 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 3.7 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 3.3) 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 3.4) 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**.

3.6.5 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.

4 Interrupt controller

The ST20-TP1 supports external interrupts, enabling an on-chip subsystem or external interrupt pin to interrupt the currently running process in order to run an interrupt handling process.

The ST20-TP1 interrupt subsystem supports eight prioritized interrupts. In addition, there is an interrupt level controller (refer to chapter 5) which multiplexes fourteen incoming interrupts onto the eight programmable interrupt levels. This multiplexing is controllable by software.

All interrupts are a higher priority than the low priority process queue. Each interrupt can be programmed to be at a lower priority or a higher priority than the high priority process queue, this is determined by the **Priority** bit in the **HandlerWptr0-7** registers.

Note: Interrupts (**Interrupt0-7**) which are specified as higher priority must be contiguous from the highest numbered interrupt downwards, i.e. 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**.

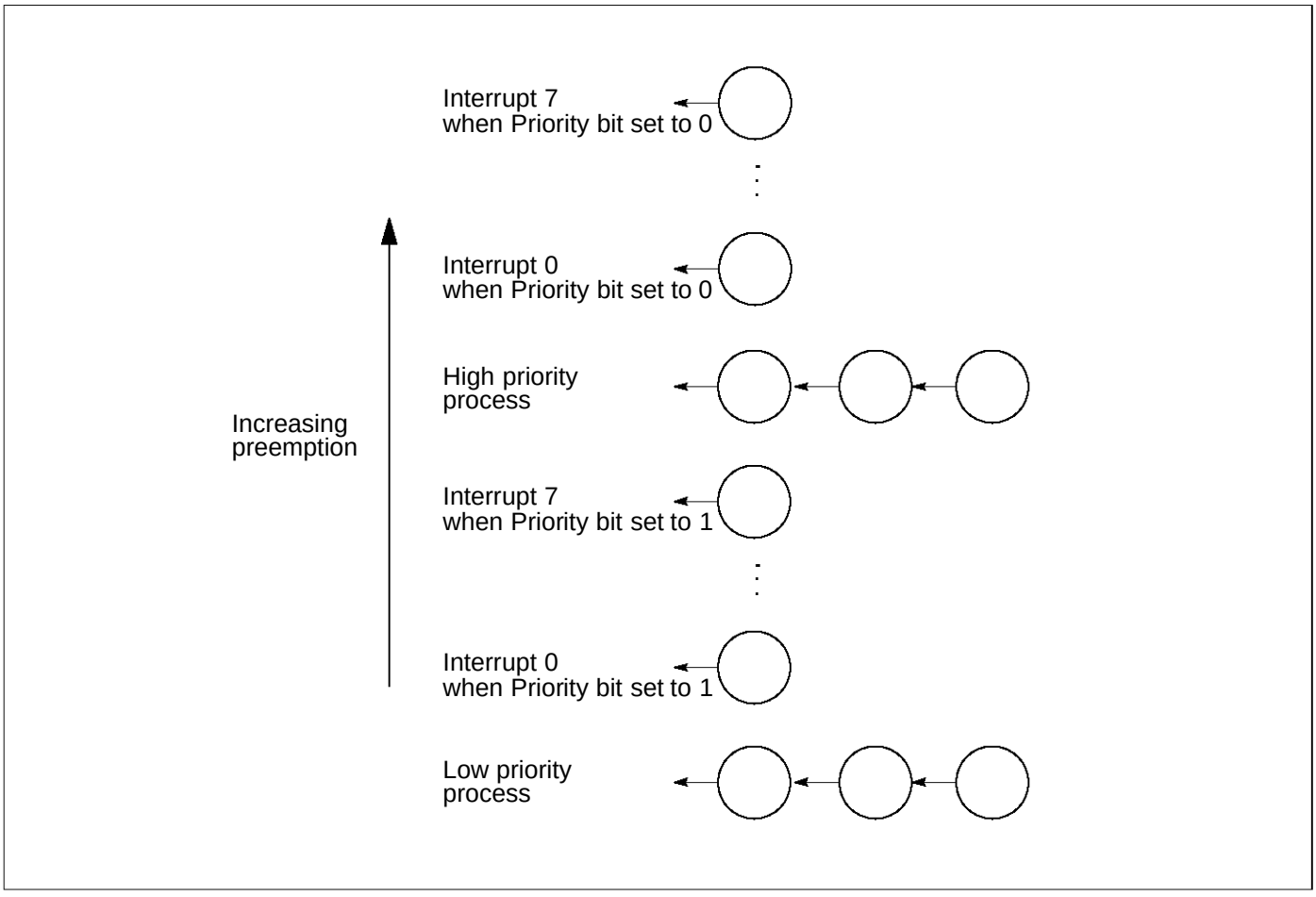


Figure 4.1 Interrupt priority

Interrupts on the ST20-TP1 are implemented via an on-chip interrupt controller peripheral. An interrupt can be signalled to the controller by one of the following:

- a signal on an external **Interrupt** pin
- a signal from an internal peripheral or subsystem
- software asserting an interrupt in a bit mask.

4.1 Interrupt vector table

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

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

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

4.2 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 workspace of the interrupt handler as shown in Figure 4.2. Each interrupt level has its own workspace.

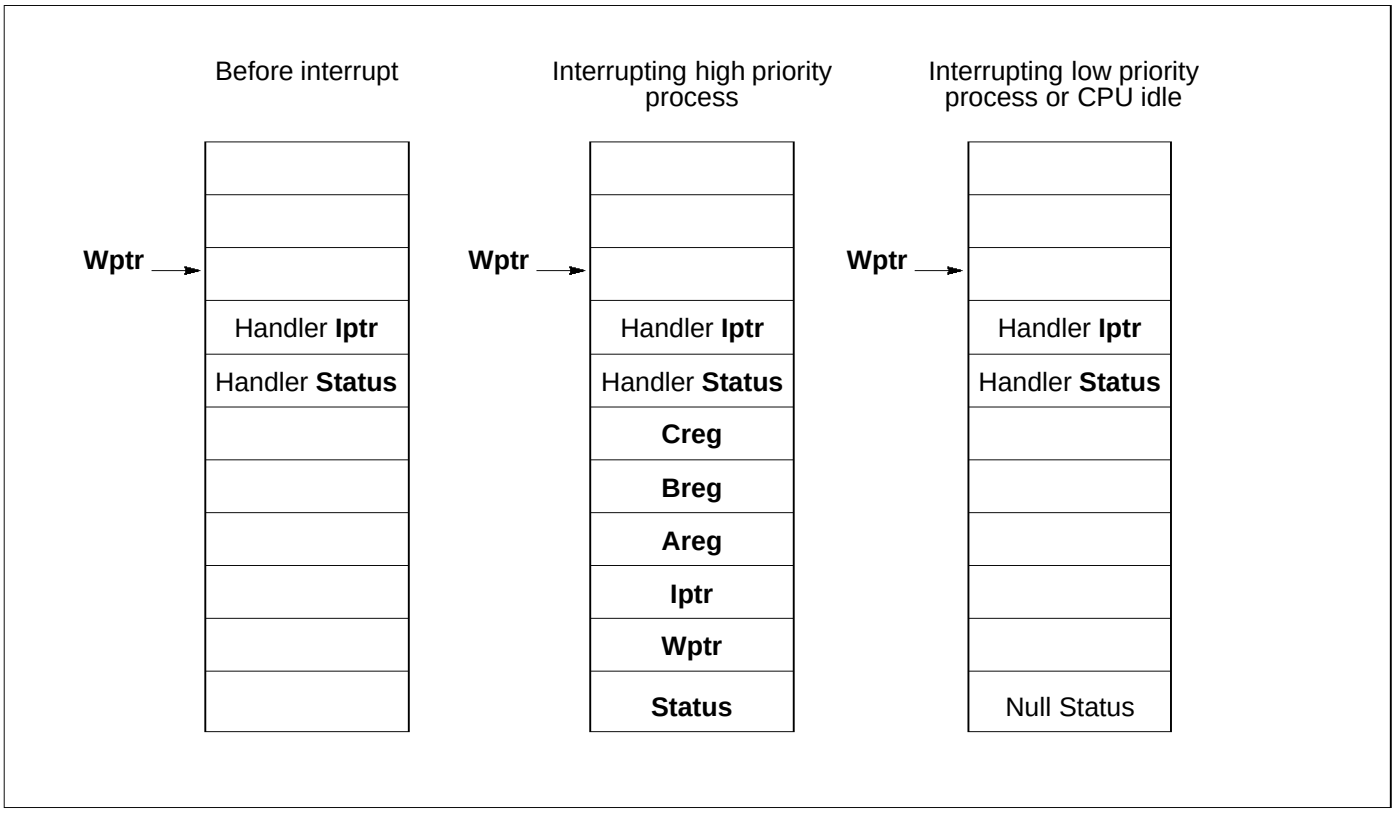


Figure 4.2 State of interrupted process

The interrupt routine is initialized with space below **Wptr**. The **Iptr** and **Status** word for the routine are stored there permanently. This should be programmed before the **Wptr** is written into the vector table. The behavior of the interrupt differs depending on the priority of the CPU when the interrupt occurs.

When an interrupt occurs when the CPU was 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 workspace. The values of **Iptr** and **Status** to be used by the interrupt handler are loaded from this workspace and starts executing the handler. The value of **Wptr** is then set to the bottom of this save area.

When an interrupt occurs when the CPU was 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 all higher priority interrupts have been serviced (see section 4.4).

Interrupts always take priority over low priority processes. When 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.

4.3 Interrupt latency

Interrupt latency is 0.5 µs when the interrupt handler and the currently executing process are both using internal memory. For external memory accesses the interrupt latency increases.

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.

4.4 Preemption 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 preempt 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.

4.5 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.

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

3 Interrupt handlers must not perform block move instructions.

4 Interrupt handlers must not cause program traps. However they may be trapped by a scheduler trap.

4.6 Interrupt configuration registers

The interrupt controller is allocated a 4k block of memory in the internal peripheral address space. Information on interrupts is stored in registers as detailed in the following section. The registers can be examined and set by the *devlw* (device load word) and *devsw* (device store word) instructions. Note, they can not be accessed using memory instructions.

HandlerWptr register

The **HandlerWptr** registers (1 per interrupt) contain a pointer to the workspace of the interrupt handler. It also contains the **Priority** bit which determines whether the interrupt is at a higher or lower priority than the high priority process queue.

Note: Before the interrupt is enabled, by writing a 1 in the **Mask** register, the user (or toolset) must ensure that there is a valid **Wptr** in the register.

HandlerWptr	**Interrupt controller base address + #00 to #1C**	**Read/Write**
Bit	**Bit field**	**Function**
0	**Priority**	Sets the priority of the interrupt. If this bit is set to 0, the interrupt is a higher priority than the high priority process queue, if this bit is 1, the interrupt is a lower priority than the high priority process queue. 0 high priority 1 low priority
31:2	**HandlerWptr**	Pointer to the workspace of the interrupt handler.
1		RESERVED. Write 0.

Table 4.1 **HandlerWptr** register format — one register per interrupt

TriggerMode register

Each interrupt channel can be programmed to trigger on rising/falling edges or high/low levels on the external **Interrupt**.

TriggerMode	**Interrupt controller base address + #40 to #5C**	**Read/Write**
Bit	**Bit field**	**Function**
2:0	**Trigger**	Control the triggering condition of the **Interrupt**, as follows: **Trigger2:0** **Interrupt triggers on** 000 No trigger mode 001 High level - triggered while input high 010 Low level - triggered while input low 011 Rising edge - low to high transition 100 Falling edge - high to low transition 101 Any edge - triggered on rising and falling edges 110 No trigger mode 111 No trigger mode

Table 4.2 **TriggerMode** register format — one register per interrupt

Note: Level triggering is different to edge triggering in that if the input is held at the triggering level, a continuous stream of interrupts is generated.

Mask register

An interrupt mask register is provided in the interrupt controller to selectively enable or disable external interrupts. This mask register also includes a global interrupt disable bit to disable all external interrupts whatever the state of the individual interrupt mask bits.

To complement this the interrupt controller also includes an interrupt pending register which contains a pending flag for each interrupt channel. The **Mask** register performs a masking function on the **Pending** register to give control over what is allowed to interrupt the CPU while retaining the ability to continually monitor external interrupts.

On start-up, the **Mask** register is initialized to zero's, thus all interrupts are disabled, both globally and individually. When a 1 is written to the **GlobalEnable** bit, the individual interrupt bits are still

disabled and must also have a 1 individually written to the **InterruptEnable** bit to enable the respective interrupt.

Mask	**Interrupt controller base address + #C0**	**Read/Write**
Bit	**Bit field**	**Function**
0	**Interrupt0Enable**	When set to 1, interrupt 0 is enabled. When 0, interrupt 0 is disabled.
1	**Interrupt1Enable**	When set to 1, interrupt 1 is enabled. When 0, interrupt 1 is disabled.
2	**Interrupt2Enable**	When set to 1, interrupt 2 is enabled. When 0, interrupt 2 is disabled.
3	**Interrupt3Enable**	When set to 1, interrupt 3 is enabled. When 0, interrupt 3 is disabled.
4	**Interrupt4Enable**	When set to 1, interrupt 4 is enabled. When 0, interrupt 4 is disabled.
5	**Interrupt5Enable**	When set to 1, interrupt 5 is enabled. When 0, interrupt 5 is disabled.
6	**Interrupt6Enable**	When set to 1, interrupt 6 is enabled. When 0, interrupt 6 is disabled.
7	**Interrupt7Enable**	When set to 1, interrupt 7 is enabled. When 0, interrupt 7 is disabled.
16	**GlobalEnable**	When set to 1, the setting of the interrupt is determined by the specific **InterruptEnable** bit. When 0, all interrupts are disabled.
15:8		RESERVED. Write 0.

Table 4.3 **Mask** register format

The **Mask** register is mapped onto two additional addresses so that bits can be set or cleared individually.

Set_Mask (address ‘interrupt base address + #C4’) allows bits to be set individually. Writing a ‘1’ in this register sets the corresponding bit in the **Mask** register, a ‘0’ leaves the bit unchanged.

Clear_Mask (address ‘interrupt base address + #C8’) allows bits to be cleared individually. Writing a ‘1’ in this register resets the corresponding bit in the **Mask** register, a ‘0’ leaves the bit unchanged.

Pending register

The **Pending** register contains a bit per interrupt with each bit controlled by the corresponding interrupt. A read can be used to examine the state of the interrupt controller while a write can be used to explicitly trigger an interrupt.

A bit is set when the triggering condition for an interrupt is met. All bits are independent so that several bits can be set in the same cycle. Once a bit is set, a further triggering condition will have no effect. The triggering condition is independent of the **Mask** register.

The highest priority interrupt bit is reset once the interrupt controller has made an interrupt request to the CPU.

The interrupt controller receives external interrupt requests and makes an interrupt request to the CPU when it has a pending interrupt request of higher priority than the currently executing interrupt handler.

Pending	**Interrupt controller base address + #80**	**Read/Write**
Bit	**Bit field**	**Function**
0	**PendingInt0**	Interrupt 0 pending bit.
1	**PendingInt1**	Interrupt 1 pending bit.
2	**PendingInt2**	Interrupt 2 pending bit.
3	**PendingInt3**	Interrupt 3 pending bit.
4	**PendingInt4**	Interrupt 4 pending bit.
5	**PendingInt5**	Interrupt 5 pending bit.
6	**PendingInt6**	Interrupt 6 pending bit.
7	**PendingInt7**	Interrupt 7 pending bit.

Table 4.4 **Pending** register format

The **Pending** register is mapped onto two additional addresses so that bits can be set or cleared individually.

Set_Pending (address ‘interrupt base address + #84’) allows bits to be set individually. Writing a ‘1’ in this register sets the corresponding bit in the **Pending** register, a ‘0’ leaves the bit unchanged.

Clear_Pending (address ‘interrupt base address + #88’) allows bits to be cleared individually. Writing a ‘1’ in this register resets the corresponding bit in the **Pending** register, a ‘0’ leaves the bit unchanged.

Note: If the CPU wants to write or clear some bits of the **Pending** register, the interrupts should be masked (by writing or clearing the **Mask** register) before writing or clearing the **Pending** register. The interrupts can then be unmasked.

Exec register

The **Exec** register keeps track of the currently executing and preempted interrupts. A bit is set when the CPU starts running code for that interrupt. The highest priority interrupt bit is reset once the interrupt handler executes a return from interrupt (*iret*).

Exec	**Interrupt controller base address + #100**	**Read/Write**
Bit	**Bit field**	**Function**
0	**Interrupt0Exec**	Set to 1 when the CPU starts running code for interrupt 0.
1	**Interrupt1Exec**	Set to 1 when the CPU starts running code for interrupt 1.
2	**Interrupt2Exec**	Set to 1 when the CPU starts running code for interrupt 2.
3	**Interrupt3Exec**	Set to 1 when the CPU starts running code for interrupt 3.
4	**Interrupt4Exec**	Set to 1 when the CPU starts running code for interrupt 4.
5	**Interrupt5Exec**	Set to 1 when the CPU starts running code for interrupt 5.
6	**Interrupt6Exec**	Set to 1 when the CPU starts running code for interrupt 6.
7	**Interrupt7Exec**	Set to 1 when the CPU starts running code for interrupt 7.

Table 4.5 **Exec** register format

The **Exec** register is mapped onto two additional addresses so that bits can be set or cleared individually.

Set_Exec (address ‘interrupt base address + #104’) allows bits to be set individually. Writing a ‘1’ in this register sets the corresponding bit in the **Exec** register, a ‘0’ leaves the bit unchanged.

Clear_Exec (address ‘interrupt base address + #108’) allows bits to be cleared individually. Writing a ‘1’ in this register resets the corresponding bit in the **Exec** register, a ‘0’ leaves the bit unchanged.

5 Interrupt level controller

The interrupt level controller extends the number of possible interrupts to fourteen.

There are fourteen interrupts generated in the ST20-TP1 system and each of these is assigned to one of the interrupt controller's eight inputs. Thus each of the interrupt controller's inputs responds to zero or more of the fourteen system interrupts.

An interrupt handler routine is able to ascertain the source of an interrupt where two or more system interrupts are assigned to one handler by doing a device read from the **InputInterrupts** register (see Table 5.2) and examining the bits that correspond to the system interrupts assigned to that handler.

The assignment of interrupts to peripherals is given in Table 29.2 on page 148.

5.1 Interrupt level controller registers

The interrupt level controller is programmable via configuration registers. These registers can be examined and set by the *devlw* (device load word) and *devsw* (device store word) instructions.

IntPriority registers

The priority assigned to each of the input interrupts is programmable via the **IntPriority** registers.

The interrupt level controller asserts interrupt output *N* when one or more of the input interrupts with programmed priority equal to *N* are high. It is level sensitive and re-timed at the input, thus incurring one cycle of latency.

IntPriority	**Interrupt level controller base address + #00 to #34**	**Read/Write**
Bit	**Bit field**	**Function**
2:0	**IntPriority**	Determines the priority of each interrupt input. **IntPriority2:0** — **Asserts output interrupt** 000 — 0 (lowest priority) 001 — 1 010 — 2 011 — 3 100 — 4 101 — 5 110 — 6 111 — 7 (highest priority)

Table 5.1 **IntPriority** register format — 1 register per interrupt

InputInterrupts register

The **InputInterrupts** register is a read only register. It contains a vector which shows all of the input interrupts, so bit 0 of the read data corresponds to **InterruptIn0**, bit 1 corresponds to **InterruptIn1**, etc.

Inputinterrupts	**Interrupt level controller base address + #38**	**Read only**
Bit	**Bit field**	**Function**
13:0	InputInterrupts	Input interrupt levels.

Table 5.2 **InputInterrupts** register format

6 Instruction set

This chapter provides information on the 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.

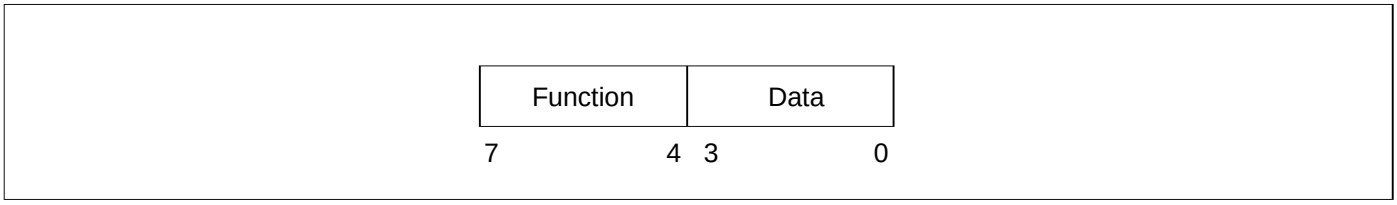


Figure 6.1 Instruction format

For further information on the instruction set refer to the *ST20 C2/C4 Instruction Set Manual (document number 72-TRN-273-01).*

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

The Primary Instructions Table 6.3 gives the basic function code. 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 instr uction 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 descheduling and error 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	7	jump	D, T
1	1X	ldlp	1	load local pointer	
2	2X	pfix	0 to 3	prefix	
3	3X	ldnl	1	load non-local	
4	4X	ldc	1	load constant	
5	5X	ldnlp	1	load non-local pointer	
6	6X	nfix	0 to 3	negative prefix	
7	7X	ldl	1	load local	
8	8X	adc	2 to 3	add constant	O
9	9X	call	8	call	
A	AX	cj	1 or 7	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 6.3 Primary functions

Memory code	Mnemonic	Processor cycles	Name	Notes
22FA	testpranal	1	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	ldprodid	1	load device identity	
27FE	ldmemstartval	1	load value of **MemStart** address	

Table 6.4 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	2	add	A, O
FC	sub	2	subtract	A, O
25F3	mul	3	multiply	A, O
27F2	fmul	5	fractional multiply	A, O
22FC	div	4 to 35	divide	A, O
21FF	rem	3 to 35	remainder	A, O
F9	gt	2	greater than	A
25FF	gtu	2	greater than unsigned	A
F4	diff	1	difference	
25F2	sum	1	sum	
F8	prod	3	product	A
26F8	satadd	2 to 3	saturating add	A
26F9	satsub	2 to 3	saturating subtract	A
26FA	satmul	4	saturating multiply	A

Table 6.5 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	1	long sum	
24FF	ldiff	1	long diff	
23F1	lmul	4	long multiply	A
21FA	ldiv	3 to 35	long divide	A, O
23F6	lshl	2	long shift left	A
23F5	lshr	2	long shift right	A
21F9	norm	3	normalize	A
26F4	slmul	4	signed long multiply	A, O
26F5	sulmul	4	signed times unsigned long multiply	A, O

Table 6.6 Long arithmetic operation codes

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

Table 6.7 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 6.8 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	1 to 7	enable timer	
22FE	dist		disable timer	I

Table 6.9 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	3 to 6	alt wait	D
24F5	altend	8	alt end	
24F9	enbs	1 to 2	enable skip	
23F0	diss	1	disable skip	
21F2	resetch	3	reset channel	
24F8	enbc	1 to 4	enable channel	
22FF	disc	1 to 6	disable channel	

Table 6.10 Input and output operation codes

Memory code	Mnemonic	Processor cycles	Name	Notes
22F0	ret	2	return	
21FB	ldpi	1	load pointer to instruction	
23FC	gajw	2 to 3	general adjust workspace	
F6	gcall	6	general call	
22F1	lend	4 to 5	loop end	T

Table 6.11 Control operation codes

Memory code	Mnemonic	Processor cycles	Name	Notes
FD	startp	5 to 6	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 6.12 Scheduling operation codes

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

Table 6.13 Error handling operation codes

Memory code	Mnemonic	Processor cycles	Name	Notes
25FB	move2dinit	1	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 6.14 2D block move operation codes

Memory code	Mnemonic	Processor cycles	Name	Notes
27F4	crcword	34	calculate crc on word	A
27F5	crcbyte	10	calculate crc on byte	A
27F6	bitcnt	3	count bits set in word	A
27F7	bitrevword	1	reverse bits in word	
27F8	bitrevnbits	2	reverse bottom n bits in word	A

Table 6.15 CRC and bit operation codes

Memory code	Mnemonic	Processor cycles	Name	Notes
27F3	cflerr	2	check floating point error	E
29FC	fptesterr	1	load value true (FPU not present)	
26F3	unpacksn	4	unpack single length floating point number	A
26FD	roundsn	7	round single length floating point number	A
26FC	postnormsn	7 to 8	post-normalize correction of single length floating point number	A
27F1	ldinf		load single length infinity	

Table 6.16 Floating point support operation codes

Memory code	Mnemonic	Processor cycles	Name	Notes
2CF7	cir	2 to 4	check in range	A, E
2CFC	ciru	2 to 4	check in range unsigned	A, E
2BFA	cb	2 to 3	check byte	A, E
2BFB	cbu	2 to 3	check byte unsigned	A, E
2FFA	cs	2 to 3	check sixteen	A, E
2FFB	csu	2 to 3	check sixteen unsigned	A, E
2FF8	xsword	2	sign extend sixteen to word	A
2BF8	xbword	3	sign extend byte to word	A

Table 6.17 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 6.18 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 6.19 Device access instructions

Memory code	Mnemonic	Processor cycles	Name	Notes
60F5	wait	4 to 10	wait	D
60F4	signal	6 to 10	signal	

Table 6.20 Semaphore instructions

Memory code	Mnemonic	Processor cycles	Name	Notes
60F0	swapqueue	3	swap scheduler queue	
60F1	swaptimer	5	swap timer queue	
60F2	insertqueue	1 to 2	insert at front of scheduler queue	
60F3	timeslice	3 to 4	timeslice	
60FC	ldshadow	6 to 23	load shadow registers	A
60FD	stshadow	5 to 17	store shadow registers	A
62FE	restart	19	restart	
62FF	causeerror	2	cause error	
61FF	iret	3 to 9	interrupt return	
2BF0	settimeslice	1	set timeslicing status	
2CF4	intdis	1	interrupt disable	
2CF5	intenb	2	interrupt enable	
2CFD	gintdis	2	global interrupt disable	
2CFE	gintenb	2	global interrupt enable	

Table 6.21 Scheduling support instructions

Memory code	Mnemonic	Processor cycles	Name	Notes
26FE	ldtraph	11	load trap handler	L
2CF6	ldtrapped	11	load trapped process status	L
2CFB	sttrapped	11	store trapped process status	S
26FF	sttraph	11	store trap handler	S
60F7	trapenb	2	trap enable	
60F6	trapdis	2	trap disable	
60FB	tret	9	trap return	

Table 6.22 Trap handler instructions

Memory code	Mnemonic	Processor cycles	Name	Notes
63F0	nop	1	no operation	

Table 6.23 No operation instructions

Memory code	Mnemonic	Processor cycles	Name	Notes
64FF	clockenb	2	clock enable	
64FE	clockdis	2	clock disable	
64FD	ldclock	1	load clock	
64FC	stclock	2	store clock	

Table 6.24 Clock instructions

7 Memory map

The ST20-TP1 processor memory has a 32-bit signed address range. Words are addressed by 30-bit word addresses and a 2-bit byte-selector identifies the bytes in the word. Memory is divided into 4 banks which can each have different memory characteristics and can be used for different purposes. In addition, part of the address range of the PI-Bus component of the interconnect system is visible via the device access instructions (see Table 6.19 on page 44).

Various memory locations at the bottom and top of memory are reserved for special system purposes. There is also a default allocation of memory banks to different uses.

7.1 System memory use

The ST20-TP1 has a signed address space where the address ranges from **MinInt** (#80000000) at the bottom to **MaxInt** (#7FFFFFFF) at the top. The ST20-TP1 has an area of 8 Kbytes of RAM at the bottom of the address space provided by on-chip memory. The bottom of this area is used to store various items of system state. These addresses should not be accessed directly but via the special instructions that allow the contents to be manipulated.

Near the bottom of the address space there is a special address **MemStart**. Memory above this address is for use by user programs while addresses below it are for private use by the processor and used for subsystem channels and trap-handlers. The address of **MemStart** can be obtained via the *ldmemstartval* instruction.

7.1.1 Subsystem channels memory

Each channel between the processor and a 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.

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.

7.1.2 Trap handlers memory

The area of memory reserved for trap handlers is broken down hierarchically. Full details on trap handlers is given in Section 3.6 on page 19

- Each high/low process priority has a set of trap handlers.
- Each set of trap handlers has a handler for each of the four trap groups (refer to Section 3.6.1 on page 20).
- Each trap group handler has a trap handler structure and a trapped process structure.
- Each of the structures contains four words, as detailed in Section 3.6.3 on page 22.

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

7.2 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 to move up to 256 bytes down in the ROM program. For large ROM images it may be necessary to encode a longer negative jump to reach the entry point of the application.

7.3 Internal peripheral space

The subsystem channels are implemented by memory mapped peripherals to the processor on-chip. These peripherals should be mapped to addresses in the top half of memory bank 2 (address range #20000000 to #40000000). They are accessed via the device memory access instructions (see Table 6.19). When used with addresses in this range the device access instructions perform a memory transaction across the command bus rather than the memory bus. This enables code, used to control off-chip peripherals, to be reused if the peripheral is re-implemented on-chip.

This area of memory is allocated to peripherals in 4K blocks, see the following memory map.

	ADDRESS	USE
BootEntry	#7FFFFFFE	Boot entry point
	↑ #40000000	User code/Data/Stack and Boot ROM
	↑ #20015000	High speed data port controller peripheral (registers accessed via CPU device accesses)
	↑ #20014000	Block move DMA controller peripheral (registers accessed via CPU device accesses)
	↑ #20013000	RESERVED
	↑ #20012000	Interrupt level controller peripheral (registers accessed via CPU device accesses)
	↑ #20011000	RESERVED
	↑ #20010000	DES descrambler controller peripheral (registers accessed via CPU device accesses)
	↑ #2000F000	MPEG1 DMA controller peripheral (registers accessed via CPU device accesses)
	↑ #2000E000	MPEG0 DMA controller peripheral (registers accessed via CPU device accesses)
	↑ #2000D000	PIO3 controller peripheral (registers accessed via CPU device accesses)
	↑ #2000C000	PIO2 controller peripheral (registers accessed via CPU device accesses)
	↑ #2000B000	PIO1 controller peripheral (registers accessed via CPU device accesses)
	↑ #2000A000	PIO0 controller peripheral (registers accessed via CPU device accesses)
	↑ #20009000	PWM and counter controller peripheral (registers accessed via CPU device accesses)
	↑ #20008000	SSC1 controller peripheral (registers accessed via CPU device accesses)
	↑ #20007000	SSC0 controller peripheral (registers accessed via CPU device accesses)
	↑ #20006000	SmartCard clock generator peripheral (registers accessed via CPU device accesses)
	↑ #20005000	ASC2 (SmartCard) controller peripheral (registers accessed via CPU device accesses)
	↑ #20004000	ASC1 controller peripheral (registers accessed via CPU device accesses)
	↑ #20003000	ASC0 controller peripheral (registers accessed via CPU device accesses)
	↑ #20002000	EMI controller peripheral (registers accessed via CPU device accesses)
	↑ #20001000	RESERVED
	↑ #20000000	Interrupt controller peripheral (registers accessed via CPU device accesses)
	↑ #00000000	External peripherals
	↑	User code/Data/Stack
MemStart	#80000140	

Figure 7.1 ST20-TP1 memory map

	ADDRESS	USE
	#80000130	Low Priority Scheduler trapped process
	#80000120	Low Priority Scheduler trap handler
	#80000110	Low Priority SystemOperations trapped process
	#80000100	Low Priority SystemOperations trap handler
	#800000F0	Low Priority Error trapped process
	#800000E0	Low Priority Error trap handler
	#800000D0	Low Priority Breakpoint trapped process
	#800000C0	Low Priority Breakpoint trap handler
	#800000B0	High Priority Scheduler trapped process
	#800000A0	High Priority Scheduler trap handler
	#80000090	High Priority SystemOperations trapped process
	#80000080	High Priority SystemOperations trap handler
	#80000070	High Priority Error trapped process
	#80000060	High Priority Error trap handler
	#80000050	High Priority Breakpoint trapped process
TrapBase	#80000040	High Priority Breakpoint trap handler
	#8000003C	RESERVED
	#80000038	HS data port DMA channel out
	#80000034	Block move DMA controller channel out
	#80000030	RESERVED
	#8000002C	Link-IC channel in
	#80000028	DES descrambler DMA channel out
	#80000024	MPEG1 DMA channel out
	#80000020	MPEG0 DMA channel out
	#8000001C	RESERVED
	#80000018	
	#80000014	
	#80000010	Link0 (boot) channel in
	#8000000C	RESERVED
	#80000008	
	#80000004	
MinInt	#80000000	Link0 Out Channel

Figure 7.1 ST20-TP1 memory map

8 Memory subsystem

The ST20-TP1 memory system consists of SRAM and a programmable memory interface. The specific details on the operation of the memory interface are described separately in Chapter 9.

The ST20-TP1 has an internal memory module of 8 Kbytes of SRAM. The internal SRAM is mapped into the base of the memory space from **MinInt** (#80000000) extending upwards, as shown in Figure 8.1.

This memory can be used to store on-chip data, stack or code for time critical routines.

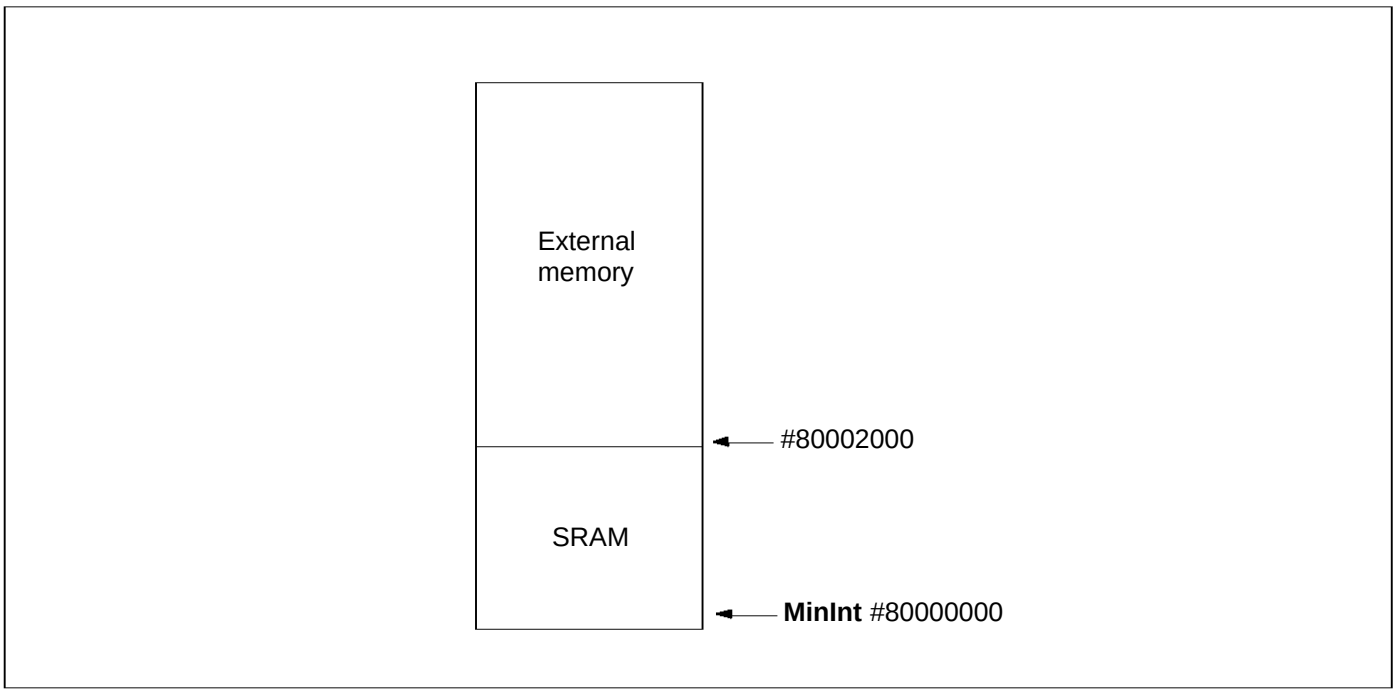


Figure 8.1 SRAM mapping

Where internal memory overlays external memory, internal memory is accessed in preference.

9 External memory interface

The External Memory Interface (EMI) controls the movement of data between the ST20-TP1 and off-chip memory.

The EMI can access a 16 Mbyte (or greater if DRAM is used) physical address space in three general purpose memory banks, and provides sustained transfer rates of up to 80 Mbytes/s for SRAM, and up to 40 Mbytes/s using page-mode DRAM. The EMI includes programmable strobes to support direct interfacing to MPEG decoder devices, and is designed to support the memory subsystems required in most set top receiver applications with zero external support logic including 16 and 32-bit DRAM devices.

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

The external address space is partitioned into four banks, with each bank occupying one quarter of the address space (see Figure 9.1). This allows the implementation of mixed memory systems, with support for DRAM, SRAM, EPROM, VRAM and I/O. The timing of each of the four memory banks can be selected separately, with a different device type being placed in each bank with no external hardware support.

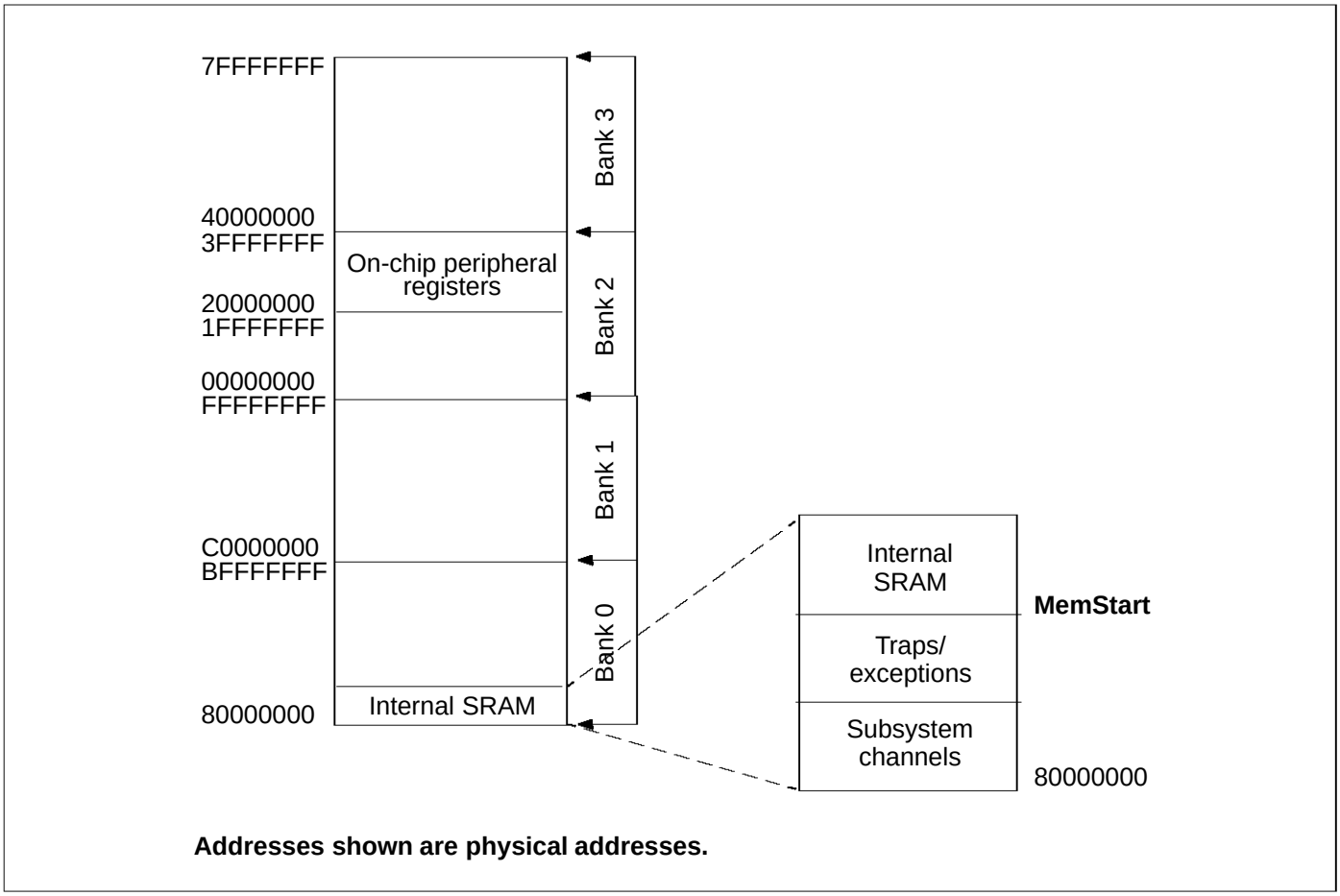


Figure 9.1 Memory allocation

On-chip internal SRAM is located at the bottom of memory. Internal SRAM is internally divided into three regions. The first at the bottom is used for channel storage space, the second region is reserved for traps and exceptions, the third region is free for program use. The boundary between the second and third region is called **MemStart** and is the lowest location in memory available for general use.

Support is provided for MPEG application devices. Bank 2 of the EMI is nominally allocated as the peripheral bank. It is in this address range that the on-chip peripheral registers appear when using device accesses. Strobes in this bank are provided to support access to the external MPEG audio and MPEG video application devices. The programmability of the EMI and the format of these strobes make the ST20-TP1 suitable for use with a range of MPEG application ICs available today and in the future.

Word addressing is used. Support for byte and part-word addressing is provided.

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

9.1 Pin functions

The following section describes the functions of the external memory interface pins. Note that a signal name prefixed by **not** indicates active low.

MemData0-31

The data bus transfers 32, 16 or 8-bit data items depending on the bus width configuration. The least significant bit of the data bus is always **MemData0**. The most significant bit varies with bus width, **MemData31** for 32-bit data items, **MemData15** for 16-bit data items, and **MemData7** for 8-bit data items.

MemAddr2-23

The address bus may be operated in both multiplexed and non-multiplexed modes. When a bank is configured to contain DRAM, or other multiplexed memory, then the internally generated 32-bit address is multiplexed as row and column addresses through the external address bus.

notMemBE0-3

The ST20-TP1 uses word addressing and four byte-enable strobes are provided. Use of the byte enable pins depends on the bus width.

- 32-bit wide memory is defined as an array of 4 byte words with 30 address bits selecting a 4 byte word. Each byte of this array is addressable with the byte enable pins **notMemBE0-3** selecting a byte within a word.
- 16-bit wide memory is defined as an array of 2 byte words with 31 address bits selecting a 2 byte word and **notMemBE0-1** selecting a byte within the word.
- 8-bit wide memory is defined as an array of 1 byte words with 32 address bits selecting a word.

For 16-bit and 8-bit wide memory, the lower order address bits (**A1** and **A0**) are multiplexed onto the unused byte-enable pins to give an address bus 31 or 32-bits wide respectively.

notMemBE0 addresses the least significant byte of a word. Both strobes have the same timing and may be configured to be active on read and or write cycles.

The function of the byte enables **notMemBE0-3** for different bank size configurations is given in Table 9.1 below. Note that other bus masters must not drive the same data pins during a write.

	External port size		
	32-bit	**16-bit**	**8-bit**
notMemBE3	enables **MemData24-31**	becomes **A1**	becomes **A1**
notMemBE2	enables **MemData16-23**	undefined	becomes **A0**
notMemBE1	enables **MemData8-15**	enables **MemData8-15**	undefined
notMemBE0	enables **MemData0-7**	enables **MemData0-7**	enables **MemData0-7**

Table 9.1 **notMemBE0-3** pins

notMemRAS0/1/3

One programmable RAS strobe is allocated to each of banks 0, 1 and 3 which are decoded on chip. If a bank is programmed to contain DRAM, or other multiplexed memory, then the associated **notMemRAS** pin acts as its RAS strobe by default. For banks which do not contain DRAM the **notMemRAS** pin is available as a general purpose programmable strobe.

notMemCAS0-3

The programmable CAS strobes can be individually programmed to be in one of two modes.

- Bank mode in which each strobe is used as the CAS strobe for a single bank.
- Byte mode in which the CAS strobe is used as a byte decoded CAS strobe and can be used across multiple banks.

Byte mode is used to support 16 or 32-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 DRAMs that have 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.

Byte mode and bank mode can be mixed in an application if the DRAM bank or banks that use byte mode are 16 bits wide. In this case only **notMemCAS0** and **notMemCAS1** need to be in byte mode and the other two CAS strobes can be used either as a bank mode CAS strobe or as a general purpose strobe.

Note, the only useful combinations of byte mode CAS strobes are all four programmed to byte mode to support 32-bit DRAM banks, and **notMemCAS0** and **notMemCAS1** programmed to byte mode to support 16-bit DRAM banks.

For banks which do not contain DRAM the **notMemCAS** pin is available as a general purpose programmable strobe.

CAS strobes in bank mode

One programmable CAS strobe is allocated to each of banks 0, 1, 2 and 3 which are decoded on chip. If a bank is programmed to contain DRAM, or other multiplexed memory, then the associated **notMemCAS** pin acts as its CAS strobe by default.

CAS strobes in byte mode

For banks containing DRAM, which require 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 the bank being accessed, 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 of the CAS strobes in this mode will go low at the start of the cycle and remain low until the end of the cycle.

The table below shows the correspondence between widest byte decoded DRAM bank size and use of byte mode strobes, and data bytes and the byte mode CAS strobes. Only the CAS strobes that enable bytes that are being accessed will be active during an access cycle.

CAS strobe	Widest byte mode DRAM bank	
	32-bit	16-bit
notMemCAS3	enables **MemData24-31**	bank 3 CAS strobe in byte mode or programmable strobe
notMemCAS2	enables **MemData16-23**	bank 2 CAS strobe in byte mode or programmable strobe
notMemCAS1	enables **MemData8-15**	enables **MemData8-15**
notMemCAS0	enables **MemData0-7**	enables **MemData0-7**

Table 9.2 Byte mode **notMemCAS0-3** strobe pins

notMemPS0/1/3

These additional general purpose programmable strobes (one for each of banks 0, 1 and 3) may be programmed in the same way as the **notMemCAS0/1/3** strobes.

notCS0-1 and notCDSTRB0-1

Four strobes are provided in bank 2 to support access to the external MPEG audio and MPEG video decoder devices. There are two decoder IC chip selects (**notCS0-1**) and two compressed data strobes (**notCDSTRB0-1**).

MemWait

Wait states can be generated by taking **MemWait** high. **MemWait** is sampled during **RASTime** and **CASTime**. **MemWait** retains the state of any strobe during the cycle in which **MemWait** was asserted. **MemWait** suspends the cycle counter and the strobe generation logic until deasserted. When **MemWait** is de-asserted cycles continue as programmed by the configuration interface.

MemReq, MemGranted

Direct memory access (DMA) can be requested at any time by driving the synchronous **MemReq** signal high. The address and data buses are tristated after the current memory access or refresh cycle terminates.

Strobes are left inactive during the DMA transfer. If a DMA is active for longer than one programmed refresh interval then external logic is responsible for providing refresh.

The **MemGranted** signal follows the timing of the bus being tristated and can be used to signal to the device requesting the DMA that it has control of the bus.

Table 9.3 below lists the processor pin state while **MemGranted** is asserted.

MemGranted asserted	
Pin name	**Pin state**
MemAddr2-23	floating
MemData0-31	floating
notMemBE0-3	inactive
notMemRAS0/1/3	inactive
notMemCAS0-3	inactive
notMemPS0/1/3	inactive
notMemRf	inactive
notMemRd	inactive
notCS0-1	inactive
notCDSTRB0-1	inactive

Table 9.3 Pin states while **MemGranted** is asserted

notMemRd

The **notMemRd** signal indicates that the current cycle is a read cycle. It is asserted low at the beginning of the read cycle and deasserted high at the end of the read cycle.

notMemRf

The **notMemRf** signal indicates that the current cycle is a refresh cycle. It is asserted low at the beginning of the refresh cycle and deasserted high at the end of the refresh cycle.

ProcClockOut

Reference signal for external bus cycles. **ProcClockOut** oscillates at the processor clock frequency.

BootSource0-1

The **BootSource0-1** pins determine whether the ST20-TP1 will boot from link or from ROM. When the **BootSource0-1** pins are both held low the ST20-TP1 will boot from its link. If either or both pins are high the ST20-TP1 will boot from ROM, as shown in Table 9.4. Boot code is run from a slow

external ROM placed in bank 3 (at the top of memory). The **BootSource0-1** pins also encode the size of bank 3. This overrides the value in the configuration registers for the **PortSize** for bank 3.

BootSource1:0	Function
0:0	Boot from link. The ST20-TP1 loads bootstrap down the link and executes from **MemStart**.
0:1	Boot from ROM. Port size of bank 3 hardwired to 32-bits.
1:0	Boot from ROM. Port size of bank 3 hardwired to 16-bits.
1:1	Boot from ROM. Port size of bank 3 hardwired to 8-bits.

Table 9.4 **BootSource0-1** pin settings

When booting from the link, the port size of bank 3 must be configured as with any other EMI parameter, otherwise the **PortSize** field in the **ConfigDataField1** register for bank 3 (see Section 9.3) will be overridden by the value on the **BootSource0-1** pins.

If the ST20-TP1 is set to boot from link, the bootstrap must execute from internal memory until the EMI has been configured. If this is not possible then the EMI must be completely configured using *poke* commands down a link before loading the bootstrap into external memory and executing it.

9.2 External bus cycles

The external memory interface is designed to provide efficient support for dynamic memory without compromising support for other devices, such as static memory and IO devices. This flexibility is provided by allowing the required waveforms to be programmed via configuration registers (see Section 9.3).

Memory is byte addressed, with words aligned on four-byte boundaries for 32-bit devices and on two-byte boundaries for 16-bit devices.

During read cycles byte level addressing is performed internally by the ST20-TP1. The EMI can read bytes, half-words or words. It always reads the size of the bank.

During read or write cycles the ST20-TP1 uses the **notMemBE0-3** 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 **MemAddr2-23**, however the low order address bits **A0** and **A1** have different functions depending on the size of the external data bus, see Table 9.1. The least significant bit of the data bus is always **MemData0**. The most significant bit can be adjusted dynamically to suit the required external bus size.

Note that data pins which are not used during a write access are tristated, for example, for an 8-bit bus pins **MemData8-31** are tristated.

A generic memory interface cycle consists of a number of defined periods, or times, as shown in Figure 9.2. This generic memory cycle uses DRAM terminology to clarify the use of the interface in the most complex situations, but can be programmed to provide waveforms for a wide range of other device types. The timing of each of the four memory banks can be programmed separately, with a different device type being placed in each bank with no external hardware support.

The **RASTime** and **CASTime** are consecutive. The **CASTime** can be followed by concurrent **Precharge** and **BusRelease** times. Thus, for DRAM, the times are used for RAS, CAS, and precharge

respectively. For non-multiplexed addressed memory the **RASTime** can be programmed to be zero.

If the **RASTime** is programmed to be non-zero, and page-mode memory is programmed in a bank, the **RASTime** will only occur if consecutive accesses are not in the same page. The **RASTime** will not commence until the **PrechargeTime** for a previous access to the same bank has completed. During the **RASTime** a transition can be programmed on the RAS and programmable strobes, but not on the CAS or byte enable strobes.

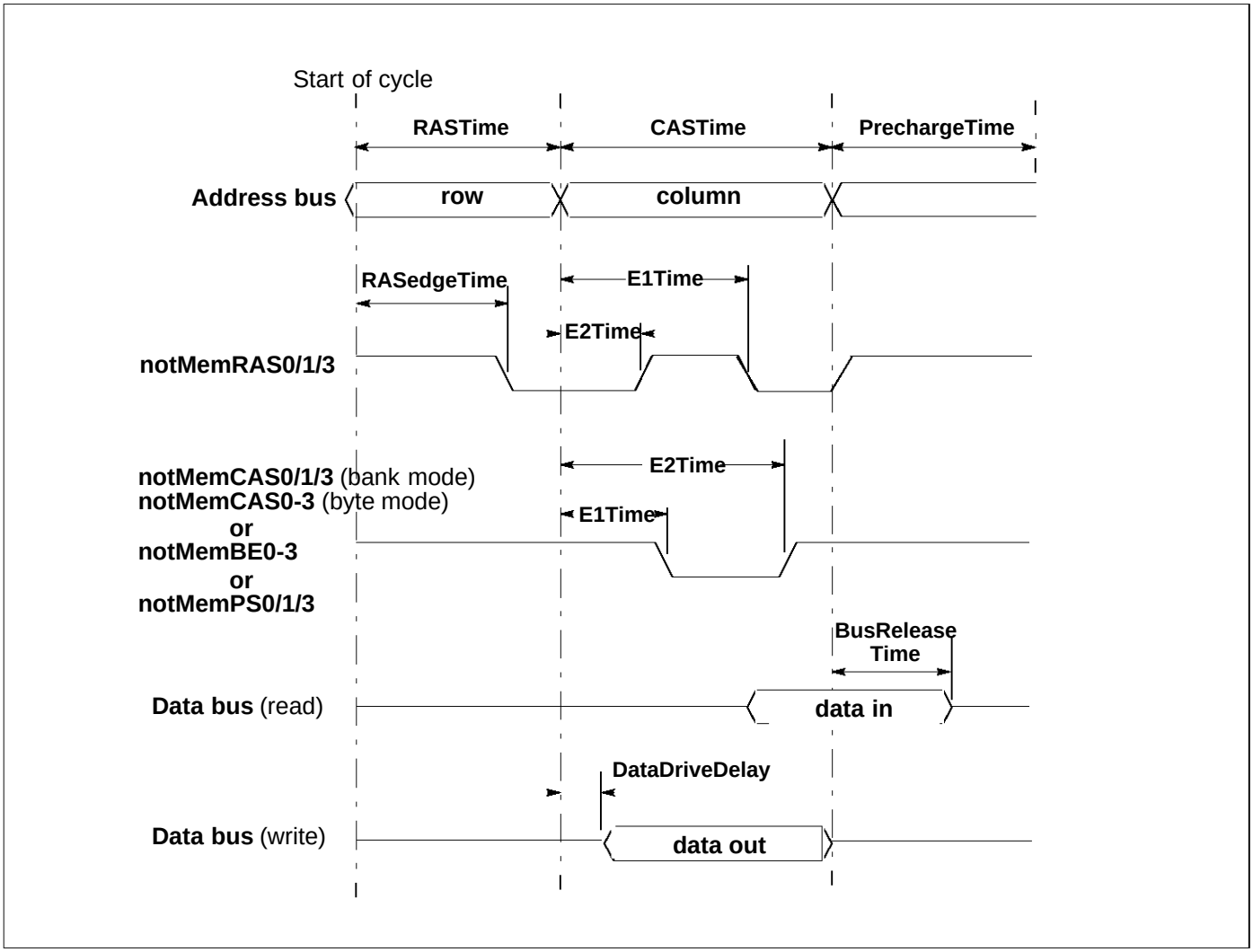


Figure 9.2 Generic memory cycle

During the **CASTime** the programmable strobes and byte-enable strobes are active. The address is output on the address bus without being shifted. Write data is valid during **CASTime**. Read data is latched into the interface on the rising edge of the internal processor clock which coincides or proceeds the programmed **notMemCAS** e2 time, as shown in Figure 9.3.

Note that the e1 and e2 times for the **notMemBE** and the **notMemCAS** strobes when in byte mode must be ≥ 2 phases.

The **PrechargeTime** and **BusReleaseTime** commence concurrently at the end of the **CASTime**. A **PrechargeTime** will occur to the current bank if:

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

The **BusReleaseTime** is provided to allow slow devices to float to a high impedance state.

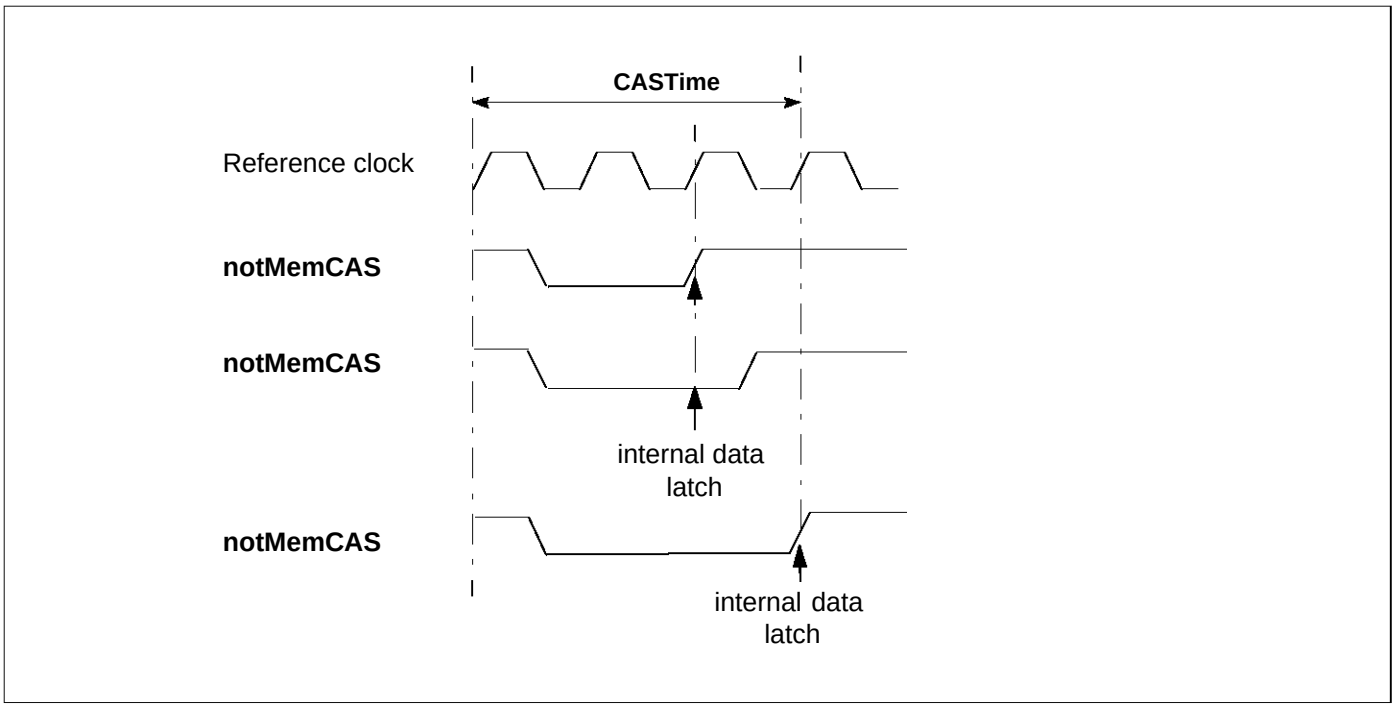


Figure 9.3 Data latching

9.2.1 Refresh

Configuration fields are provided which specify the banks which require refreshing and the interval between successive refreshes.

The EMI ensures that **notMemCAS** and **notMemRAS** 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**.**

The behavior of the **notMemCAS** strobes during a refresh cycle is dependent on the programming of the byte mode configuration field.

In bank mode the **notMemCAS** strobe is taken low at the beginning of the refresh time. The position of the RAS falling edge (**RASedge**) and the time before **notMemRAS** and **notMemCAS** can be taken high again (**RefreshTime**) are programmable**.** Each of these actions occurs in sequence

for each bank. A cycle is inserted between each bank in order to spread current peaks. If no DRAM has been programmed for a bank then no transitions occur on the RAS or CAS strobes.

In byte mode all of the **notMemCAS** strobes in byte mode are taken low at the beginning of the refresh time for bank0. The position of the RAS falling edge (**RASedge**) and the time before **notMemRAS** strobe can be taken high again (**RefreshTime**) are programmable**.** The **notMemRAS** strobes for each of the banks is taken low in sequence. A cycle is inserted between each bank in order to spread current peaks. If no DRAM has been programmed for a bank then no transitions occur on the **RAS** or **CAS** strobes.

Note: No refreshes take place unless a **DRAMinitialize** command in the **ConfigCommand** register (see Section 9.3.1 on page 65) is performed.

Note, only CAS_before_RAS refresh is supported.

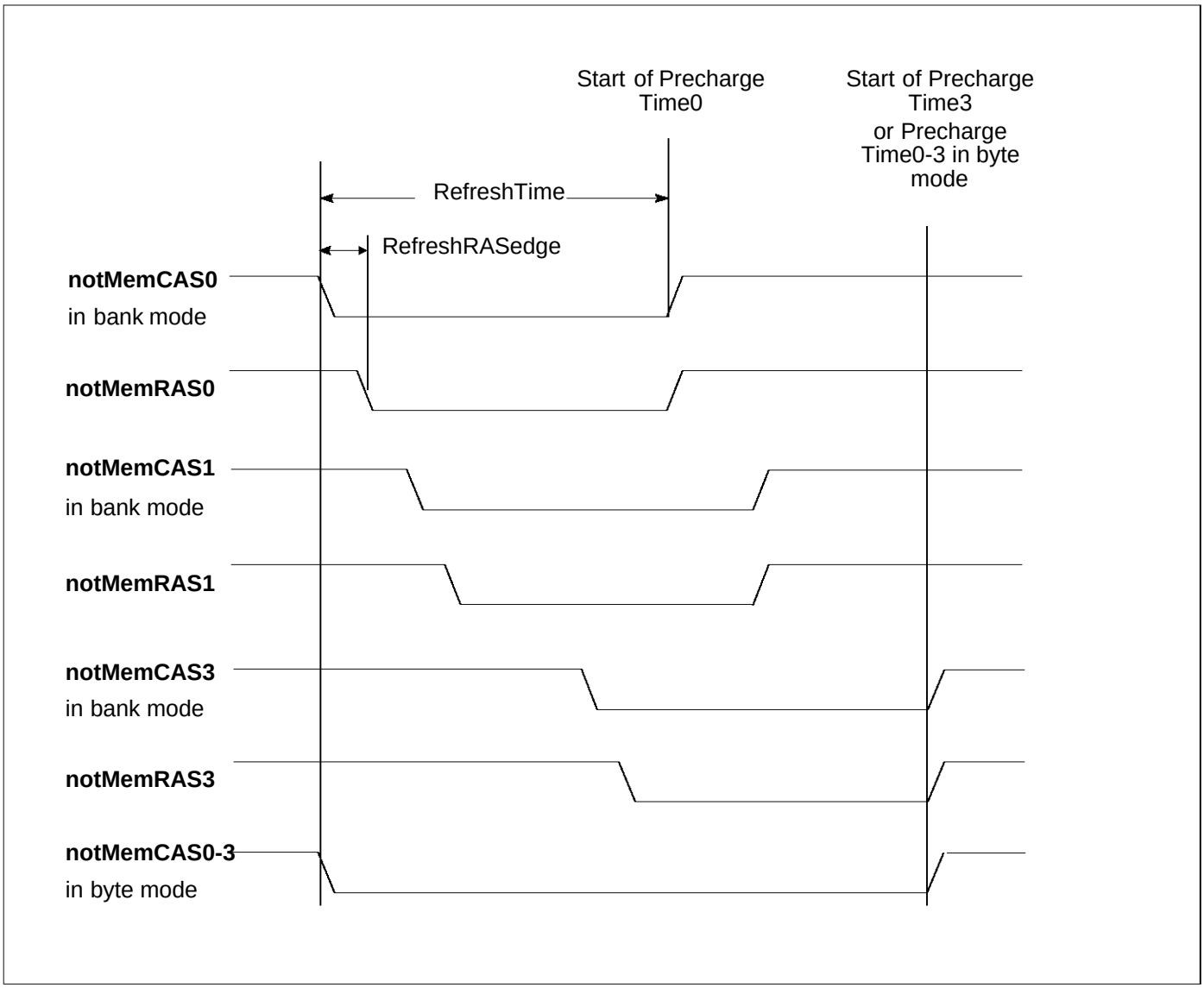


Figure 9.4 Refresh

9.2.2 Wait

MemWait is provided so that external cycles can be extended to enable variable access times (for example, shared memory access). **MemWait** is sampled on a rising clock edge before being passed into the EMI. It is only effective when the EMI is in the RAS or CAS times and has the effect of holding the RAS and CAS counter values 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 will not be inhibited, but transitions on the rising and falling edges of the cycle after will not occur. Figure 9.5 and Figure 9.6 show the extension of the external memory cycle and the delaying of strobe transitions.

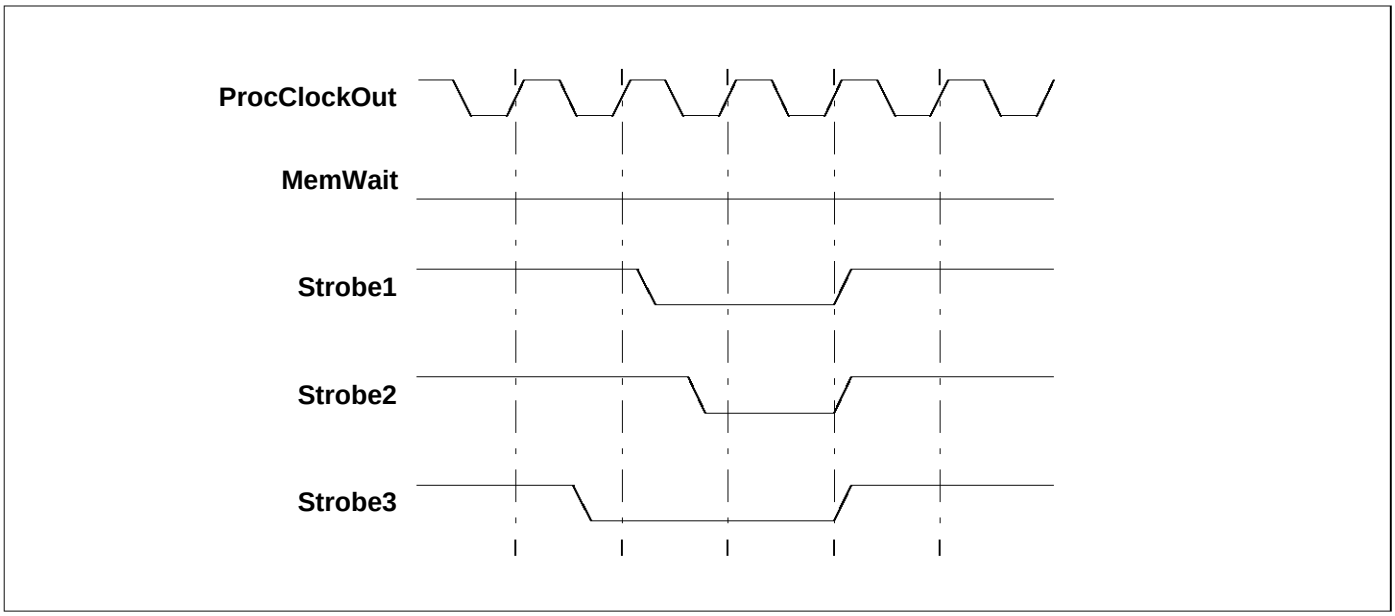


Figure 9.5 Strobe activity without **MemWait**

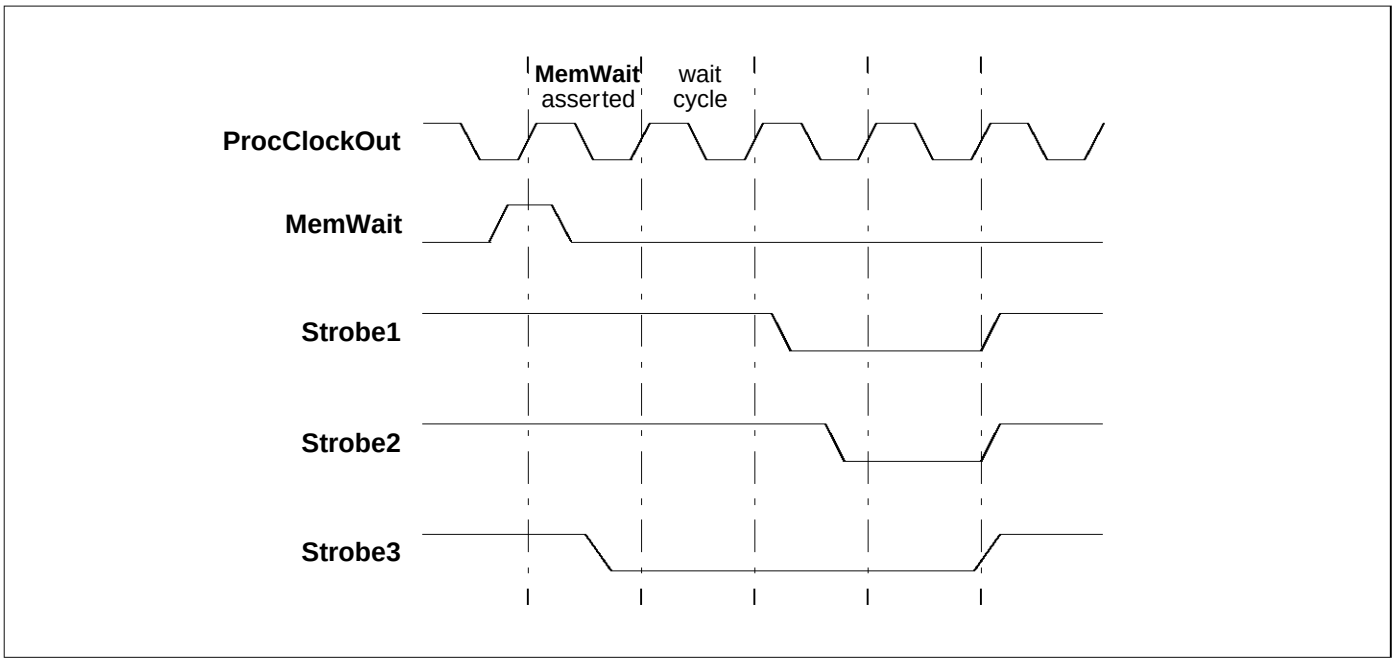


Figure 9.6 Strobe activity with **MemWait**

9.2.3 Support for MPEG application devices

Bank 2 of the EMI is nominally allocated as the peripheral bank. It is in this address range that the on-chip peripheral registers appear when using device accesses to memory. Strobes in this bank are provided to support access to the external MPEG audio and MPEG video application devices.

Four strobes are provided in bank 2. There are two MPEG decoder IC chip selects (**notCS0-1**) and two decoder compressed data strobes (**notCDSTRB0-1**).

Note, the **notMemRAS** and **notMemPS** strobes are *not* provided in bank 2. The **notMemCAS2** strobe is provided to support 32-bit wide DRAM banks in byte mode.

A single set of programmable timing and configuration parameters are provided for bank 2. The timings are different however for the **notCS0-1** strobes as one to four wait states are inserted after the first clock cycle by an internal wait state generator. The number of wait states is programmed by the values of the **MemAddr14-15** bits during the access according to Table 9.5. The wait signal from the internal wait state generator is ORed with the external **MemWait** pin so additional wait states may be added to any external access in the bank2 address range. The wait states for the **notCS0-1** strobes may be removed by disabling the **MemWait** pin in the configuration register for bank2.

MemAddr15	MemAddr14	Wait states
0	0	1
0	1	2

Table 9.5 Wait states for **notCS0-1** accesses

MemAddr15	MemAddr14	Wait states
1	0	3
1	1	4

Table 9.5 Wait states for **notCS0-1** accesses

The **notCS0-1** and **notCDSTRB0-1** strobes are active for different parts of the bank address range as detailed in Table 9.6 below.

Address range	Active strobe
#00000000 - #00000FFF - 1 wait state, #00004000 - #00004FFF - 2 wait states, #00008000 - #00008FFF - 3 wait states, #0000C000 - #0000CFFF - 4 wait states	**notCS0**
#00001000 - #00001FFF - 1 wait state, #00005000 - #00005FFF - 2 wait states, #00009000 - #00009FFF - 3 wait states, #0000D000 - #0000DFFF - 4 wait states	**notCS1**
#00002000 - #00002FFF - no wait states	**notCDSTRB0**
#00003000 - #00003FFF - no wait states	**notCDSTRB1**

Table 9.6 Strobe activity in bank 2

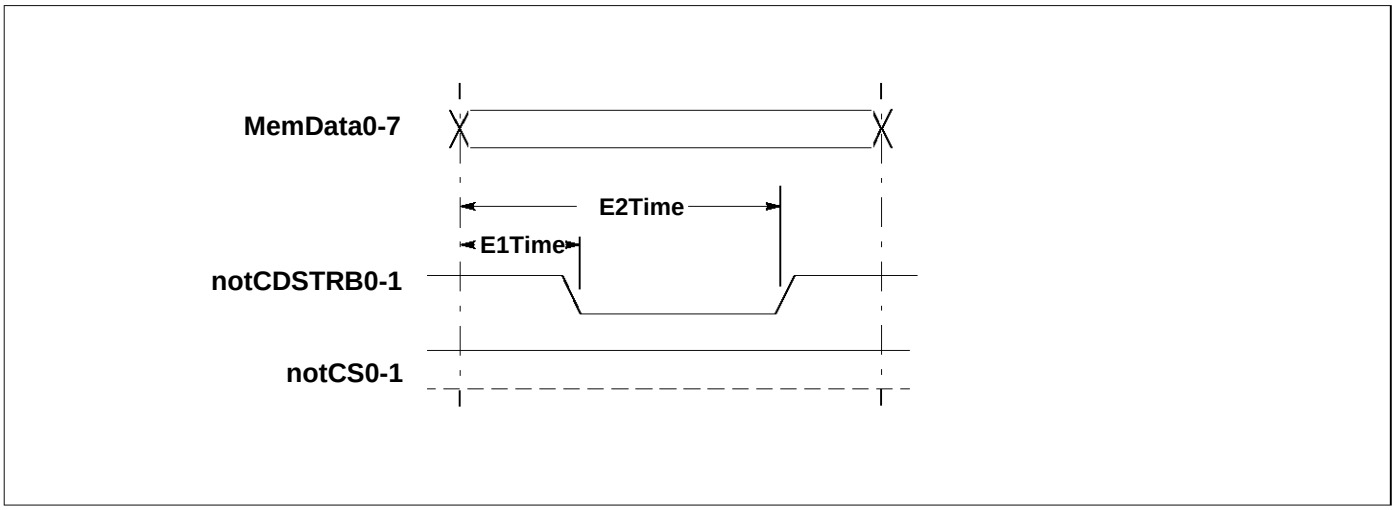


Figure 9.7 Compressed data write cycle - bank 2

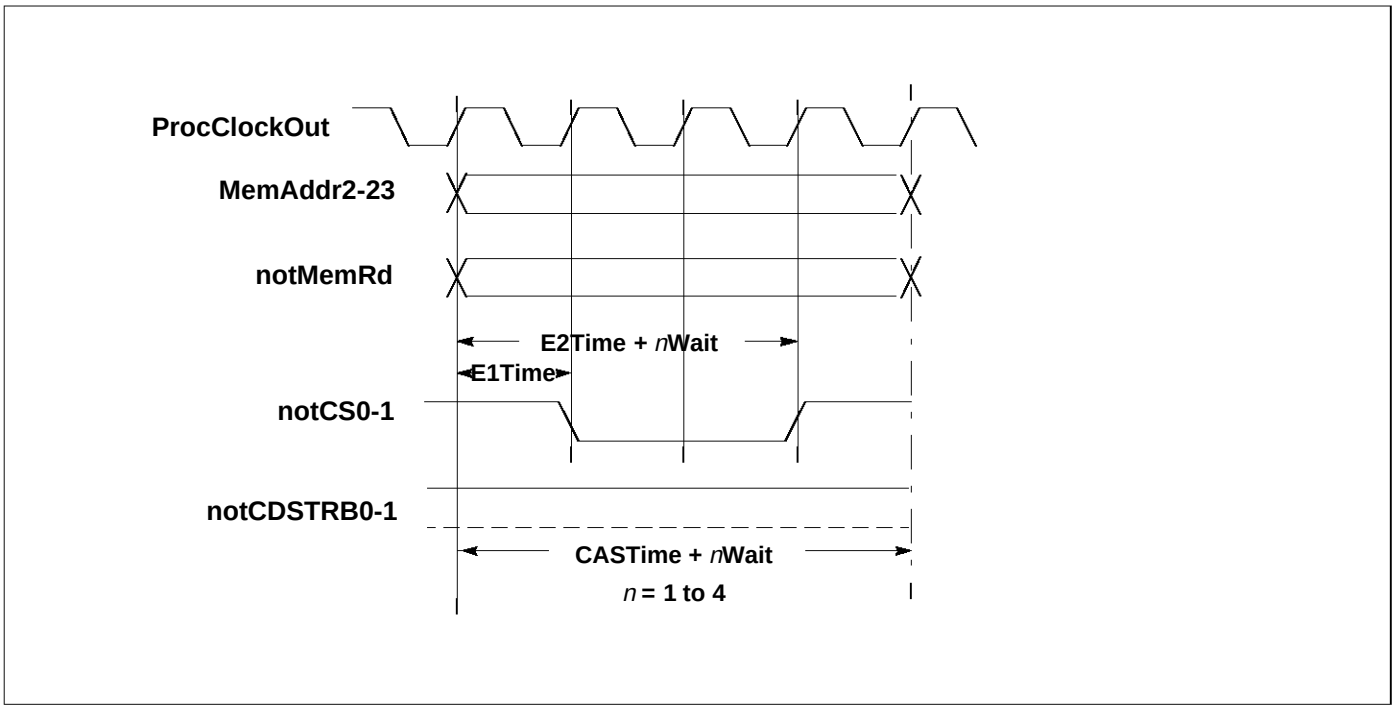


Figure 9.8 Register read/write cycle - bank 2

9.3 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.

The EMI has four banks of four 32-bit configuration registers to set up the four EMI banks. In addition there is another register to set the pad drive strength. For safe configuration each of the four banks must be configured in a single operation in cooperation with the EMI control logic. To enable this, there is a bank of four temporary registers (**ConfigDataField0-3**) inside the EMI configuration logic which can be filled with an entire bank before being transferred in a single operation to the EMI. The data is only transferred when the EMI is able to receive it. This single operation is the **WriteConfig** command in the **ConfigCommand** register. A typical configuration sequence is to program each individual temporary register (**ConfigDataField0-3**) followed by a write to the **WriteConfig** address to transfer the data to the EMI.

The configuration logic contains six registers which are used to transfer data to and from the EMI configuration registers, as listed in Table 9.7. The registers can be examined and set by the *devlw* (device load word) and *devsw* (device store word) instructions. Note, they can not be accessed using memory instructions. These registers may be accessed independently of EMI activity, unless the configuration controller is processing a previous command, for example a **WriteConfig**.

The base address for the EMI configuration registers are given in the ST20-TP1 Memory Map, see Figure 7.1 on page 48.

Note: The EMI configuration registers can not be accessed directly, they can only be accessed via the temporary registers in the configuration logic.

Register	Address	Data byte	Read/Write	Command
ConfigCommand	**EMI base address +** #10	#00	Write	ReadConfig bank 0
		#04	Write	ReadConfig bank 1
		#08	Write	ReadConfig bank 2
		#0C	Write	ReadConfig bank 3
		#10	Write	ReadConfig **PadDriveReg**
		#20	Write	DRAMinitialize
		#40	Write	WriteConfig bank 0
		#44	Write	WriteConfig bank 1
		#48	Write	WriteConfig bank 2
		#4C	Write	WriteConfig bank 3
		#50	Write	WriteConfig **PadDriveReg**
		#60	Write	LockConfig
ConfigDataField0	**EMI base address +** #00	-	Read/Write	
ConfigDataField1	**EMI base address +** #04	-	Read/Write	
ConfigDataField2	**EMI base address +** #08	-	Read/Write	
ConfigDataField3	**EMI base address +** #0C	-	Read/Write	
ConfigStatus	**EMI base address +** #20	-	Read	

Table 9.7 EMI configuration register addresses

9.3.1 ConfigCommand register

The **ConfigCommand** register is a write only register. When a write is performed to this register, plus the associated data byte, various operations are performed as detailed in Table 9.8.

To avoid further EMI activity occurring between successive update requests, all parameters for a bank must be changed in a single operation by performing a **WriteConfig** command.

The timing information for DRAM refresh is distributed amongst access timing information in the **ConfigDataField0-3** registers. DRAM is initialized by performing a **DRAMinitialize** command. The **DRAMinitialize** command also enables refreshes to take place. If no **DRAMinitialize** command is performed no refreshes will take place.

Note, the **DRAMinitialize** command should only be written when there is DRAM in the system.

ConfigCommand	EMI base address + #10	Write only
Data byte	**Bit field**	**Function**
01000000 for bank 0 01000100 for bank 1 01001000 for bank 2 01001100 for bank 3 01010000 for PadDrive	**WriteConfig**	Transfers the contents of the **ConfigDataField0-3** into the specified bank in the EMI configuration registers. All parameters for a specified bank are changed in one atomic action, to avoid further EMI activity occurring between successive update requests.
00000000 for bank 0 00000100 for bank 1 00001000 for bank 2 00001100 for bank 3 00010000 for PadDrive	**ReadConfig**	Copies the contents of the specified bank in the EMI configuration registers into **ConfigDataField0-3**.
00100000	**DRAMinitialize**	Initialize any DRAM in the system.
01100000	**LockConfig**	Disables the **WriteConfig** and **DRAMinitialize** commands and locks the **ConfigDataField0-3** to prevent further writes.

Table 9.8 **ConfigCommand** register

9.3.2 ConfigStatus register

The **ConfigStatus** register is a read only register and contains information on whether the **ConfigDataField0-3** registers have been write locked and shows which EMI banks have been written.

ConfigStatus	EMI base address + #20	Read only
Bit	**Bit field**	**Function**
0	**WrittenBank0**	Bank 0 has been configured by the **WriteConfig** command.
1	**WrittenBank1**	Bank 1 has been configured by the **WriteConfig** command.
2	**WrittenBank2**	Bank 2 has been configured by the **WriteConfig** command.
3	**WrittenBank3**	Bank 3 has been configured by the **WriteConfig** command.
4	**WrittenPadDriveReg**	The **PadDrive** register has been written by the **WriteConfig** command.
5	**WriteLock**	**ConfigDataField0-3** registers are write locked.
31:5		Reserved

Table 9.9 **ConfigStatus** register

9.3.3 ConfigDataField0-3 registers

The bit format and functionality of the **ConfigDataField0-3** registers for transfers to/from each of the register banks are described in the following sections.

The **ConfigDataField0-3** registers are grouped, with one group of four registers containing all the information necessary to program an external bank. The format of bits in the registers depends on which EMI bank is being configured, see Figure 9.9

Figure 9.9 **ConfigDataField0-3** registers

9.3.4 Format of the data registers for transfers to/from register bank 0

This section gives the format of the **ConfigDataField0-3** registers for transfers to/from register bank 0.

ConfigDataField0 format for transfers to/from register bank 0

The **ConfigDataField0** register is a 32 bit register which can be set to read only via the **ConfigCommand** register.

The **RASbits31:2** field is a 30 bit address mask which defines which address bits are compared to determine whether a page hit has occurred. Generally it will be loaded with a field of 1's padded out by 0's.

For example, if bank 0 contained 4 Mbyte DRAM, organized as four 4 Mbit x 8 devices for a 32-bit wide interface, there would be 1 MWords of DRAM, with 1024 pages each containing 1024 words. It is necessary for **RASbits31:30** to be set to '11' to enable bank switches to be detected. The **RASbits** field for bank 0 would be:

RASbits31:2 11111111111111111110000000000

For example, for a 16-bit wide interface, the **RASbits** field for bank 0 would be:

RASbits31:2 11111111111111111111000000000

ConfigDataField0	**EMI base address + #00**	**Read/Write**
Bit	**Bit field**	**Function**
1	**PageMode**	Page mode valid
31:2	**RASbits31:2**	Defines the RAS bits in the address which should be compared to the last access to the same bank to determine whether a page hit has occurred.
0		Reserved

Table 9.10 **ConfigDataField0** format for transfers to/from register bank 0

ConfigDataField1 format for transfers to/from register bank 0

The **ConfigDataField1** register is a 32 bit register which can be set to read only via the **ConfigCommand** register.

ConfigDataField1	**EMI base address + #04**		**Read/Write**
Bit	**Bit field**	**Function**	**Units**
1:0	**PortSize**	Bit width of the bank (8,16, or 32-bits). **PortSize1:0** **Bank width** 00 Invalid 01 32-bits 10 16-bits 11 8-bits	
6:2	**ShiftAmount**	Defines how many bits to shift the bank address in order to convert it to a row address for multiplexed-addressed memory during RAStime. It is irrelevant at all other times.	
8	**MemWaitEnable**	Enables the **MemWait** pin.	
9	**RAStimeEqZero**	No RAS cycle will occur. The bank is considered to be an SRAM bank.	
10	**PrechargeTimeEqZero**	No Precharge Time will occur.	
11	**RefreshTime0**	Refresh time 0. The refresh time is a 4-bit value. **RefreshTime** bits 1, 2 and 3 are specified in **ConfigDataField1** for transfers to/from register banks 1, 2 and 3 respectively.	Cycles
15:12	**PrechargeTime**	Duration of precharge time.	Cycles
16	**TP1Enable**	MUST WRITE 1.	
21:18	**DRAM3:0**	Defines which banks require refresh.	
23:22	**RAStime**	Duration of RAS sub-cycle.	Cycles
27:24	**BusReleaseTime**	Duration of bus release time.	Cycles
31:28	**CAStime**	Duration of CAS sub-cycle.	Cycles
17, 7		Reserved	

Table 9.11 **ConfigDataField1** format for transfers to/from register bank 0

ConfigDataField2 format for transfers to/from register bank 0

The **ConfigDataField2** register is a 32 bit register which can be set to read only via the **ConfigCommand** register.

Each of the strobes (**notMemRAS**, **notMemCAS**, **notMemPS**, **notMemBE**) edges may be configured to be active during reads and/or writes, or to be inactive. The coding of the active bits is given in Table 9.12.

Active bit settings	**Strobe activity**
00	Inactive
01	Active during read only
10	Active during write only
11	Active during read and write

Table 9.12 Active bit settings

ConfigDataField2	EMI base address + #08		Read/Write
Bit	**Bit field**	**Function**	**Units**
1:0	**BEe1active**	Cycle type in which falling (E1) edge of **notMemBE** is active.	
2	**BEe1LSB**	Specifies the phase when the falling (E1) edge of **notMemBE** will occur.	
3	**DataDriveDelay0**	This is a 2-bit value (**DataDriveDelay1** is in bit 7). It is the drive delay of the data bus, as follows: **DataDriveDelay1:0** — **Drive delay of data bus** 00 — 0 phases 01 — 1 phase 10 — 2 phases 11 — 3 phases	Phases
5:4	**BEe2active**	Cycle type in which **notMemBE** rising (E2) edge is active.	
6	**BEe2LSB**	Specifies the phase when the rising (E2) edge of **notMemBE** will occur.	
7	**DataDriveDelay1**	This is a 2-bit value (**DataDriveDelay0** is in bit 3). It is the drive delay of the data bus.	Phases
9:8	**PSe1active**	Cycle type in which falling (E1) edge of **notMemPS** is active.	
10	**PSe1LSB**	Specifies the phase when the falling (E1) edge of **notMemPS** will occur.	
11	**ByteModeEnable**	Set to 1 to enable byte mode on **notMemCAS**	
13:12	**PSe2active**	Cycle type in which rising (E2) edge of **notMemPS** is active.	
14	**PSe2LSB**	Specifies the phase when the rising (E2) edge of **notMemPS** will occur.	
17:15	**RASedgeTime**	Delay from start of RAS sub-cycle to falling edge of RAS strobe.	Phases
18	**RASe1LSB**	Specifies the phase when the falling (E1) edge of **notMemRAS** will occur.	
20:19	**RASe1active**	Cycle type in which falling (E1) edge of **notMemRAS** is active.	
21	**RASe2LSB**	Specifies the phase when the rising (E2) edge of **notMemRAS** will occur.	
23:22	**RASe2active**	Cycle type in which rising (E2) edge of **notMemRAS** is active.	
25:24	**RASedgeActive**	Cycle type in which an edge of **notMemRAS** is active.	
27:26	**CASe1active**	Cycle type in which falling (E1) edge of **notMemCAS** is active.	
28	**CASe1LSB**	Specifies the phase when the falling (E1) edge of **notMemCAS** will occur.	
29	**CASe2LSB**	Specifies the phase when the rising (E2) edge of **notMemCAS** will occur.	
31:30	**CASe2active**	Cycle type in which rising (E2) edge of **notMemCAS** is active.	

Table 9.13 **ConfigDataField2** format for transfers to/from register bank 0

Note that the e1 and e2 times for the **notMemBE** and the **notMemCAS** strobes when in byte mode must be ≥2 phases.

ConfigDataField3 format for transfers to/from register bank 0

The **ConfigDataField3** register is a 32 bit register which can be set to read only via the **ConfigCommand** register.

ConfigDataField3	**EMI base address + #0C**		**Read/Write**
Bit	**Bit field**	**Function**	**Units**
3:0	**BEe1timeMSB**	The number of complete cycles from **CASTime** start to **notMemBE** falling (E1) edge.	Cycles
7:4	**BEe2timeMSB**	The number of complete cycles from **CASTime** start to **notMemBE** rising (E2) edge.	Cycles
11:8	**PSe1timeMSB**	The number of complete cycles from **CASTime** start to **notMemPS** falling (E1) edge.	Cycles
15:12	**PSe2timeMSB**	The number of complete cycles from **CASTime** start to **notMemPS** rising (E2) edge.	Cycles
19:16	**RASe1timeMSB**	The number of complete cycles from **CASTime** start to **notMemRAS** falling (E1) edge.	Cycles
23:20	**RASe2timeMSB**	The number of complete cycles from **CASTime** start to **notMemRAS** rising (E2) edge.	Cycles
27:24	**CASe1timeMSB**	The number of complete cycles from **CASTime** start to **notMemCAS** falling (E1) edge.	Cycles
31:28	**CASe2timeMSB**	The number of complete cycles from **CASTime** start to **notMemCAS** rising (E2) edge.	Cycles

Table 9.14 **ConfigDataField3** format for transfers to/from register bank 0

Note that the e1 and e2 times for the **notMemBE** and the **notMemCAS** strobes when in byte mode must be ≥2 phases.

9.3.5 Format of the data registers for transfers to/from register bank 1

This section gives the format of the **ConfigDataField0-3** registers for transfers to/from register bank 1.

ConfigDataField0/2/3 format for transfers to/from register bank 1

The **ConfigDataField0**, **ConfigDataField2** and **ConfigDataField3** registers have the same format for transfers to/from register bank 1 as those given for transfers to/from register bank 0, see Table 9.10, Table 9.13 and Table 9.14 in Section 9.3.4.

ConfigDataField1 format for transfers to/from register bank 1

This register contains refresh information.

ConfigDataField1	**EMI base address + #04**		**Read/Write**
Bit	**Bit field**	**Function**	**Units**
1:0	**PortSize**	Bit width of the bank (8, 16, or 32-bits). **PortSize1:0** — **Bank width** 00 — Invalid 01 — 32-bits 10 — 16-bits 11 — 8-bits	
6:2	**ShiftAmount**	Defines how many bits to shift the bank address in order to convert it to a row address for multiplexed-addressed memory during RAS-time. It is irrelevant at all other times.	
8	**MemWaitEnable**	Enables the **MemWait** pin.	
9	**RAStimeEqZero**	No RAS cycle will occur. The bank is considered to be an SRAM bank.	
10	**PrechargeTimeEqZero**	No Precharge Time will occur.	
11	**RefreshTime1**	Refresh time 1. The refresh time is a 4-bit value. **RefreshTime** bits 0, 2 and 3 are specified in **ConfigDataField1** for transfers to/from register banks 0, 2 and 3 respectively.	Cycles
15:12	**PrechargeTime**	Duration of precharge time.	Cycles
21:17	**RefreshRASedgeTime**	Refresh RAS falling edge.	Phases
23:22	**RAStime**	Duration of RAS sub-cycle.	Cycles
27:24	**BusReleaseTime**	Duration of bus release time.	Cycles
31:28	**CAStime**	Duration of CAS sub-cycle.	Cycles
16, 7		Reserved	

Table 9.15 **ConfigDataField1** format for transfers to/from register bank 1

9.3.6 Format of the data registers for transfers to/from register bank 2

This section gives the format of the **ConfigDataField0-3** registers for transfers to/from register bank 2.

The **ConfigDataField0** register is RESERVED for transfers to/from register bank 2.

ConfigDataField1 format for transfers to/from register bank 2

This register contains refresh information.

The 12-bit refresh interval is spread across two register fields, see Table 9.19.

ConfigDataField1	**EMI base address + #04**		**Read/Write**
Bit	**Bit field**	**Function**	**Units**
1:0	**PortSize**	Bit width of the bank (8, 16, or 32-bits). **PortSize1:0** — **Bank width** 00 — Invalid 01 — 32-bits 10 — 16-bits 11 — 8-bits	
7	**BusRelMax0**	This is a 2-bit value (**BusRelMax1** is bit 7 of **ConfigDataField1** for bank 3, refer Table 9.19) which encodes a pointer to the EMI bank with the greatest BusRelease time. This BusRelease time will be inserted when the EMI is coming out of a DMA transaction. The encodings are as follows: **BusRelMax1:0** — **Greatest BusRelease time** 00 — Bank 0 01 — Bank 1 10 — Bank 2 11 — Bank 3	
8	**MemWaitEnable**	Enables the **MemWait** pin.	
11	**RefreshTime2**	Refresh time 2. The refresh time is a 4-bit value. **RefreshTime** bits 0, 1 and 3 are specified in **ConfigDataField1** for transfers to/from register banks 0, 1 and 3 respectively.	Cycles
21:16	**RefreshInterval5:0**	This is a 12-bit value (**RefreshInterval11:6** is bits 21:16 of **ConfigDataField1** for bank 3, refer Table 9.19) which defines the DRAM refresh interval between successive refreshes.	Cycles
27:24	**BusReleaseTime**	Duration of bus release time.	Cycles
31:28	**CAStime**	Duration of CAS sub-cycle.	Cycles
6:2, 10:9, 15:12, 23:22		Reserved	

Table 9.16 **ConfigDataField1** format for transfers to/from register bank 2

ConfigDataField2 format for transfers to/from register bank 2

ConfigDataField2	EMI base address + #08		Read/Write
Bit	**Bit field**	**Function**	**Units**
1:0	**BEe1active**	Cycle type in which falling (E1) edge of **notMemBE** is active.	
2	**BEe1LSB**	Specifies the phase when the falling (E1) edge of **notMemBE** will occur.	
3	**DataDriveDelay0**	This is a 2-bit value (**DataDriveDelay1** is in bit 7). It is the drive delay of the data bus, as follows: **DataDriveDelay1:0** — **Drive delay of data bus** 00 — 0 phases 01 — 1 phase 10 — 2 phases 11 — 3 phases	Phases
5:4	**BEe2active**	Cycle type in which **notMemBE** rising (E2) edge is active.	
6	**BEe2LSB**	Specifies the phase when the rising (E2) edge of **notMemBE** will occur.	
7	**DataDriveDelay1**	This is a 2-bit value (**DataDriveDelay0** is in bit 3). It is the drive delay of the data bus.	Phases
11	**ByteModeEnable**	Set to 1 to enable byte mode on **notMemCAS**	
27:26	**CASe1active**	Cycle type in which falling (E1) edge of **notMemCAS** is active.	
28	**CASe1LSB**	Specifies the phase when the falling (E1) edge of **notMemCAS** will occur.	
29	**CASe2LSB**	Specifies the phase when the rising (E2) edge of **notMemCAS** will occur.	
31:30	**CASe2active**	Cycle type in which rising (E2) edge of **notMemCAS** is active.	
10:8, 25:12		Reserved	

Table 9.17 **ConfigDataField2** format for transfers to/from register bank 2

Note that the e1 and e2 times for the **notMemBE** and the **notMemCAS** strobes when in byte mode must be ≥2 phases.

ConfigDataField3 format for transfers to/from register bank 2

ConfigDataField3	EMI base address + #0C		Read/Write
Bit	**Bit field**	**Function**	**Units**
3:0	**BEe1timeMSB**	The number of complete cycles from **CASTime** start to **notMemBE** falling (E1) edge.	Cycles
7:4	**BEe2timeMSB**	The number of complete cycles from **CASTime** start to **notMemBE** rising (E2) edge.	Cycles
27:24	**CASe1timeMSB**	The number of complete cycles from **CASTime** start to **notMemCAS** falling (E1) edge.	Cycles
31:28	**CASe2timeMSB**	The number of complete cycles from **CASTime** start to **notMemCAS** rising (E2) edge.	Cycles
23:8		Reserved	

Table 9.18 **ConfigDataField3** format for transfers to/from register bank 2

9.3.7 Format of the data registers for transfers to/from register bank 3

This section gives the format of the **ConfigDataField0-3** registers for transfers to/from register bank 3.

ConfigDataField0/2/3 format for transfers to/from register bank 3

The **ConfigDataField0**, **ConfigDataField2** and **ConfigDataField3** registers have the same format for transfers to/from register bank 3 as those given for transfers to/from register bank 0, see Table 9.10, Table 9.13 and Table 9.14 in Section 9.3.4.

ConfigDataField1 format for transfers to/from register bank 3

This register contains refresh information. The 12-bit refresh interval value is spread across two register fields.

ConfigDataField1	**EMI base address + #04**		**Read/Write**
Bit	**Bit field**	**Function**	**Units**
1:0	**PortSize**	Bit width of the bank (8,16, or 32-bits). **PortSize1:0** — **Bank width** 00 — Invalid 01 — 32-bits 10 — 16-bits 11 — 8-bits	
6:2	**ShiftAmount**	Defines how many bits to shift the bank address in order to convert it to a row address for multiplexed-addressed memory during RAS-time. It is irrelevant at all other times.	
7	**BusRelMax1**	This is a 2-bit value (**BusRelMax0** is bit 7 of **ConfigDataField1** for bank 2, refer Table 9.16) which encodes a pointer to the EMI bank with the greatest BusRelease time. This BusRelease time will be inserted when the EMI is coming out of a DMA transaction. The encodings are as follows: **BusRelMax1:0** — **Greatest BusRelease time** 00 — Bank 0 01 — Bank 1 10 — Bank 2 11 — Bank 3	
8	**MemWaitEnable**	Enables the **MemWait** pin.	
9	**RAStimeEqZero**	No RAS cycle will occur. The bank is considered to be an SRAM bank.	
10	**PrechargeTimeEqZero**	No Precharge Time will occur.	
11	**RefreshTime3**	Refresh time 3. The refresh time is a 4-bit value. **RefreshTime** bits 0, 1 and 2 are specified in **ConfigDataField1** for transfers to/from register banks 0, 1 and 2 respectively.	Cycles
15:12	**PrechargeTime**	Duration of precharge time.	Cycles
21:16	**RefreshInterval11:6**	This is a 12-bit value (**RefreshInterval5:0** is bits 21:16 of **ConfigDataField1** for bank 2, refer Table 9.16) which defines the DRAM refresh interval between successive refreshes.	Cycles
23:22	**RAStime**	Duration of RAS sub-cycle.	Cycles
27:24	**BusReleaseTime**	Duration of bus release time.	Cycles
31:28	**CAStime**	Duration of CAS sub-cycle.	Cycles

Table 9.19 **ConfigDataField1** format for transfers to/from register bank 3

9.3.8 Format of the data registers for transfers to/from PadDrive register

This final group of registers consists of just one register. The **ConfigDataField0-2** registers are reserved. The **ConfigDataField3** register is used for the pad drive strength register.

This register sets the drive strength of the EMI pads. Once locked the strength is static. Each of the address, data and strobe pads has four possible drive strengths which may be configured as given in Table 9.20.

Drive bit settings	Drive strength level	Drive strength
00	level 0	Weakest
01	level 1	↓
10	level 2	↓
11	level 3	Strongest

Table 9.20 Drive bit settings

The **PadDrive** register has fields which apply to groups of pads so that the edge rates may be tuned to reduce electrical noise or optimize speed. Also the **ProcClockOut** pin can be disabled in order to reduce power, this is the default on reset.

ConfigDataField3	**EMI base address + #0C**	**Read/Write**
Bit	**Bit field**	**Function**
1:0	**RCP0**	Drive strength of pads **notMemRAS0**, **notMemCAS0**, **notMemPS0**
3:2	**RCP1**	Drive strength of pads **notMemRAS1**, **notMemCAS1**, **notMemPS1**
5:4	**RCP2**	Drive strength of pads **notMemCAS2**, **notCS0**, **notCS1**, **notCDSTRB0**, **notCDSTRB1**
7:6	**RCP3**	Drive strength of pads **notMemRAS3**, **notMemCAS3**, **notMemPS3**
9:8	**BE1**	Drive strength of pads **notMemBE1**
11:10	**BE2**	Drive strength of pads **notMemBE2**
13:12	**A2-8**	Drive strength of pads **MemAddr2-8**, **notMemBE0**, **notMemBE3**
15:14	**A9-12**	Drive strength of pads **MemAddr9-12**
17:16	**A13-16**	Drive strength of pads **MemAddr13-16**
19:18	**A17-20**	Drive strength of pads **MemAddr17-20**
21:20	**A21-23**	Drive strength of pads **MemAddr21-23**
25:24	**D0-7**	Drive strength of pads **MemData0-7**
27:26	**D8-15**	Drive strength of pads **MemData8-15**
29:28	**D16-31**	Drive strength of pads **MemData16-31**
31	**ProcClockEnable**	When 1, **ProcClockOut** pin enabled. When 0 (default state on reset), the **ProcClock-Out** pin is disabled, thus reducing power.
23:22, 30		Reserved

Table 9.21 **ConfigDataField3** format for transfers to/from **PadDrive** register

9.4 EMI initialization

9.4.1 Reset

When the EMI is reset, the configuration register file loads a default set of parameters suitable for running boot code from a slow external ROM placed in bank 3 (at the top of memory). The refresh interval is reset to zero and no refresh requests are generated until this parameter is changed and the **DRAMinitialize** command is issued to the configuration logic.

The **WriteLock** bit in the **ConfigStatus** register is cleared to enable new parameters to be configured by software.

9.4.2 Bootstrap

When external reset is removed, the ST20-TP1 will start to execute bootstrap code from the area of memory determined by the setting of the **BootSource0-1** pins (see Table 9.4 on page 56).

If the ST20-TP1 is set to boot from a link, the bootstrap must execute from internal memory until the EMI has been configured. If this is not possible, the EMI must be completely configured using *poke* operations (see Section 10.2.3 on page 80) down the link before loading the bootstrap into external memory and executing it.

9.4.3 Initializing DRAM banks

The timing information for DRAM refresh is spread over the configuration registers (**ConfigDataField0-3**). DRAM initialization is performed by an explicit command (**DRAMinitialize** command in the **ConfigCommand** register) once the configuration is loaded. This command causes 8 consecutive refresh transactions to occur.

Default configuration

The default configuration is loaded into all four banks on reset. The parameters shown in Table 9.22 are also set in the configuration registers.

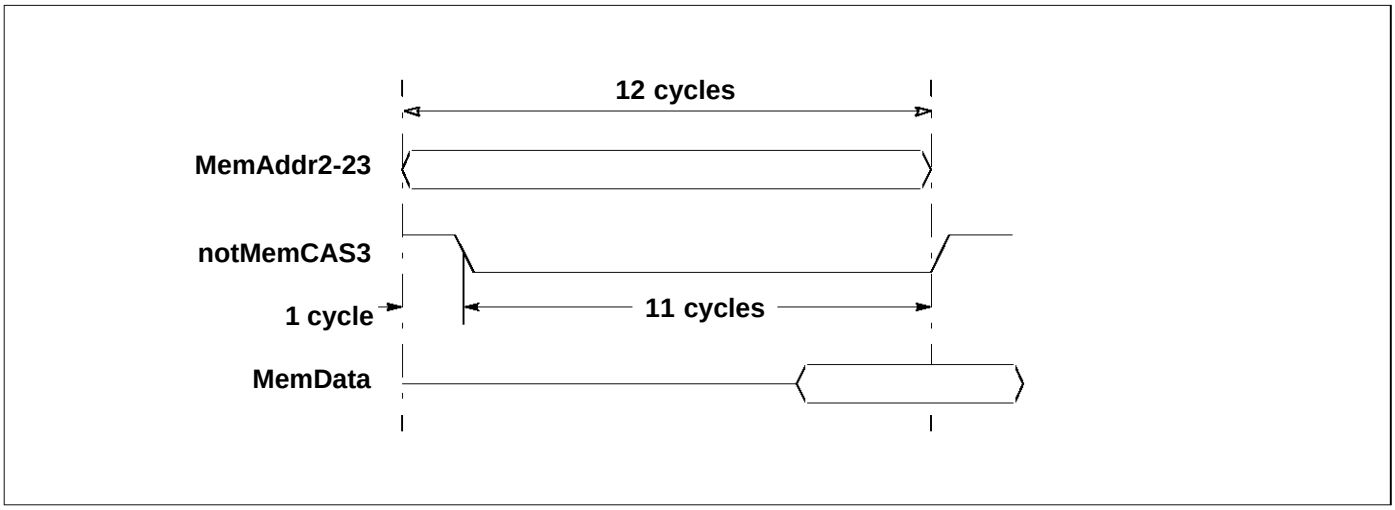


Figure 9.10 Timing of default access

Parameter	Default value
RASbits31:2	#0 (all banks)
PageMode	Cleared (all banks)
PortSize	Value on **BootSource0-1** pin
ShiftAmount	0 (all banks)
BusReleaseMax1:0	3
MemWaitEnable	Set (all banks)
RAStimeEqZero	Set (all banks)
PrechargeTimeEqZero	Set (all banks)
RefreshTime0,1,2,3	Cleared
PrechargeTime	0 (all banks)
DRAM3:0	All cleared
RefreshRASedgeTime	0
RefreshInterval	0
RAStime	0 (all banks)
BusReleaseTime	3 cycles (all banks)
CAStime	12 cycles (all banks)
RAS, BE strobes	Inactive (all banks)
CAS, PS e1 and e2 active	Only on reads (all banks)
CASe1 time	2 phases
CASe2 time	24 phases
PSe1 time	0 phases
PSe2 time	24 phases
DataDriveDelay1:0	2 phases (all banks)
PadDriveStrength	All 0, weakest drive strength
ProcClockEnable	**ProcClockOut** pin enabled
ByteModeEnable	Byte mode disabled

Table 9.22 Default parameters

10 System services

The system services module includes all the necessary logic to initialize and sustain operation of the device and also includes error handling and analysis facilities.

10.1 Reset and Analyse

The ST20-TP1 has 3 pins to support reset and analyse: **notRST**, **CPUReset** and **CPUAnalyse**.

10.1.1 Power-on-Reset

notRST provides a ”hard” reset function and must be asserted (low) before the clocks and power are stable, but should only be de-asserted (high) *after* the clocks and power are stable to guarantee well-defined behavior.

When **notRST** is asserted (regardless of any other inputs), all modules are asynchronously forced into their power-on reset state.

When **notRST** is de-asserted the CPU enters its boot sequence which can either be in off-chip ROM or can be received down a link (see Section 10.2 on bootstrap). The rising edge of **notRST** is internally synchronized and delayed until the clocks are stable before this sequence starts.

Note: **notTRST** (TAP Reset) must have been asserted before **notRST** is de-asserted.

10.1.2 Soft Reset

During the Power-on-Reset, the entire chip is affected, including the Clock Control Logic, which take a long time to reset. An alternative, ”soft” reset is provided which does not affect the clocks, and takes a lot less time. This form of reset must only be used when the system is up and running, i.e. *not* on power-up.

Soft Reset is invoked by taking **CPUReset** high when **CPUAnalyse** is low, provided **notRST** is de-asserted.

10.1.3 Analyse

If **CPUAnalyse** is taken high when the ST20-TP1 is running, the ST20-TP1 will halt at the next descheduling point. **CPUReset** may then be asserted. When **CPUReset** comes low again the ST20-TP1 will be in its reset state, but the previous memory configuration and several status flags and register values will be maintained, permitting analysis of the halted machine.

An input OS-link will continue with outstanding transfers. An output OS-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 ST20-TP1 halting.

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

10.1.4 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 ST20-TP1 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 ST20-TP1 can alternatively be set to halt-on-error to prevent further corruption and allow postmortem debugging. The ST20-TP1 also supports user defined trap handlers, see Section 3.6 on page 19 for details.

If a high priority process preempts 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 preempted 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.

10.2 Bootstrap

The ST20-TP1 can be bootstrapped from external ROM, internal ROM or from a link. This is determined by the setting of the **BootSource0-1** pins, see Table 9.4 on page 56. If both **BootSource0-1** pins are held low it will boot from a link. If either or both pins are held high, it will boot from ROM. This is sampled once only by the ST20-TP1, before the first instruction is executed after reset.

10.2.1 Booting from ROM

When booting from ROM, the ST20-TP1 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.

10.2.2 Booting from link

When booting from a link, the ST20-TP1 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 ST20-TP1 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.

10.2.3 Peek and poke

Any location in internal or external memory can be interrogated and altered when the ST20-TP1 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.

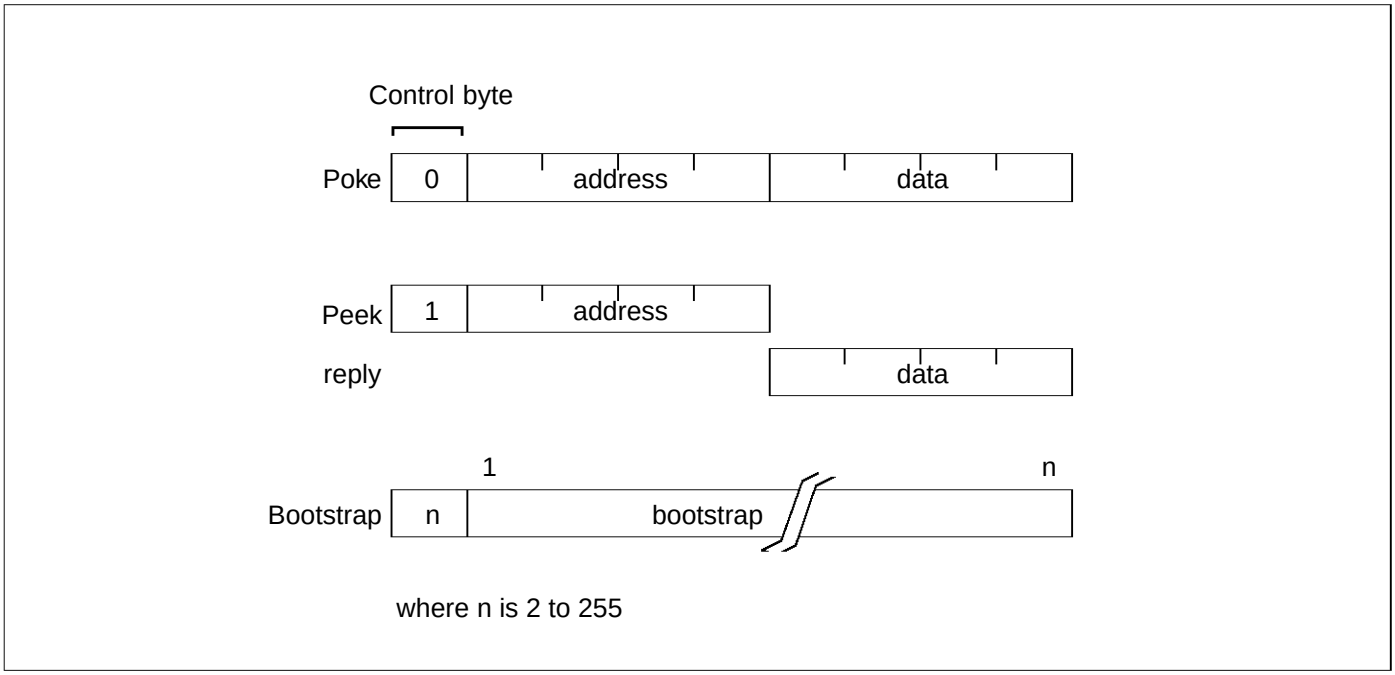


Figure 10.1 Peek, poke and bootstrap

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 ST20-TP1 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 ST20-TP1 will commence reading its bootstrap program.

11 Test access port

The ST20-TP1 Test Access Port (TAP) conforms to IEEE standard 1149.1.

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 four defined public instructions, see Table 11.1. All other instruction codes are reserved.

Instruction code [a]					**Instruction**	**Selected register**
0	0	0	0	0	EXTEST	Boundary-Scan
0	0	0	1	0	IDCODE	Identification
0	0	0	1	1	SAMPLE/PRELOAD	Boundary-Scan
1	1	1	1	1	BYPASS	Bypass

Table 11.1 Instruction codes

a. MSB ... LSB; LSB closest to **TDO.**

There are three test data registers; **Bypass**, **Boundary-Scan** and **Identification**. These registers operate according to 1149.1. The operation of the **Boundary-Scan** register is defined in the BSDL description.

11.1 Boundary scan description

This is defined for the device in a standard BSDL file. This file can be obtained through your local SGS-THOMSON distributor or sales office.

12 Clocks

An on-chip phase locked loop (PLL) generates all the internal high frequency clocks. The PLL is used to generate the internal clock frequencies needed for the CPU and the Link. Alternatively a direct clock input can provide the system clocks.

The single clock input (**ClockIn**) must be 27 MHz for PLL operation.

The ST20-TP1 can be set to operate in **TimesOneMode**, which is when the PLL is bypassed. During **TimesOneMode** the input clock must be in the range 0 to 40 MHz and should be nominally 50/50 mark space ratio.

12.1 Processor speed select

The speed of the internal processor clock is variable in discrete steps. The clock rate at which the ST20-TP1 runs is determined by the logic levels applied on the two speed select lines **SpeedSelect0-1** as detailed in Table 12.1. The frequency of **ClockIn** (fclk) for the speeds given in the table is 27 MHz.

Speed Select1:0	Processor clock speed MHz	Processor cycle time ns	Phase lock loop factor (PLLx)	High priority timer MHz	Low priority timer MHz	Link speed Mbits/s
00	TimesOneMode					
01	33.3	30.03	1.23	1.040	0.01625	19.98
10	39.96†	25.02	1.48	0.999	0.01561	19.98
11	49.95	20.02	1.85	1.040	0.01625	19.98

Table 12.1 Processor speed selection

Note: Inclusion of a speed selection in this table does not imply availability.

†Clock duty cycle is 40:60.

13 UART interface (ASC)

The UART interface, also referred to as the Asynchronous Serial Controller (ASC), provides serial communication between the ST20-TP1 and other microcontrollers, microprocessors or external peripherals.

The 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**).

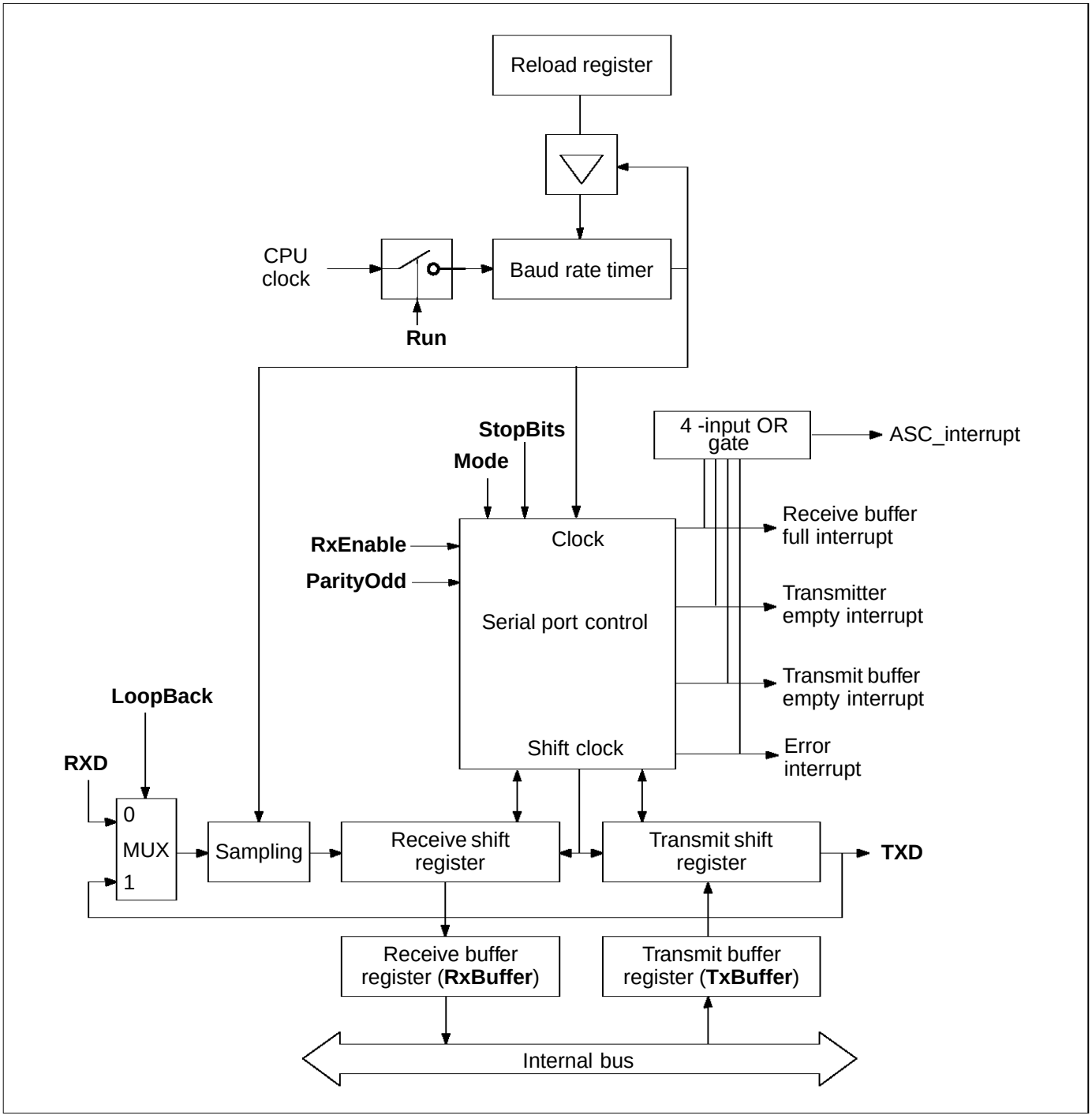


Figure 13.1 Block diagram of the ASC

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 mech-

anism to distinguish address from data bytes is included. Testing is supported by a loop-back option. A 16-bit baud rate generator provides the ASC with a separate serial clock signal.

The ASC can be set to operate in SmartCard mode for use when interfacing to a SmartCard.

13.1 Asynchronous serial controller operation

The operating mode of the serial channel ASC is controlled by the control register (**ASCControl**). This register contains control bits for mode and error check selection, and status flags for error identification.

A transmission is started by writing to the transmit buffer register (**ASCTxBuffer**), see Table 13.3.

Data transmission is double-buffered, therefore 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.

Data reception is enabled by the receiver enable bit (**RxEnable**) in the **ASCControl** register. After reception of a character has been completed the received data, and received parity bit if selected, can be read from the receive buffer register (**ASCRxBuffer**), refer to Table 13.4.

Data reception is double-buffered, so the reception of a second character may begin before the previously received character has been read out of the receive buffer register. The overrun error status flag (**OverrunError**) in the status register (**ASCStatus**), see Table 13.7, 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 **ASCStatus** 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.

13.1.1 Data frames

Data frames are selected by the setting of the **Mode** bit field in the **ASCControl** register, see Table 13.5.

8-bit data frames

8-bit data frames consist of:

- eight data bits **D0-7**;
- seven data bits **D0-6** plus an automatically generated parity bit.

Parity may be odd or even, depending on the **ParityOdd** bit in the **ASCControl** register. An even parity bit will be set, if the modulo-2-sum of the seven data bits is 1. An odd parity bit will be cleared in this case. 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 **ASCRxBuffer** register.

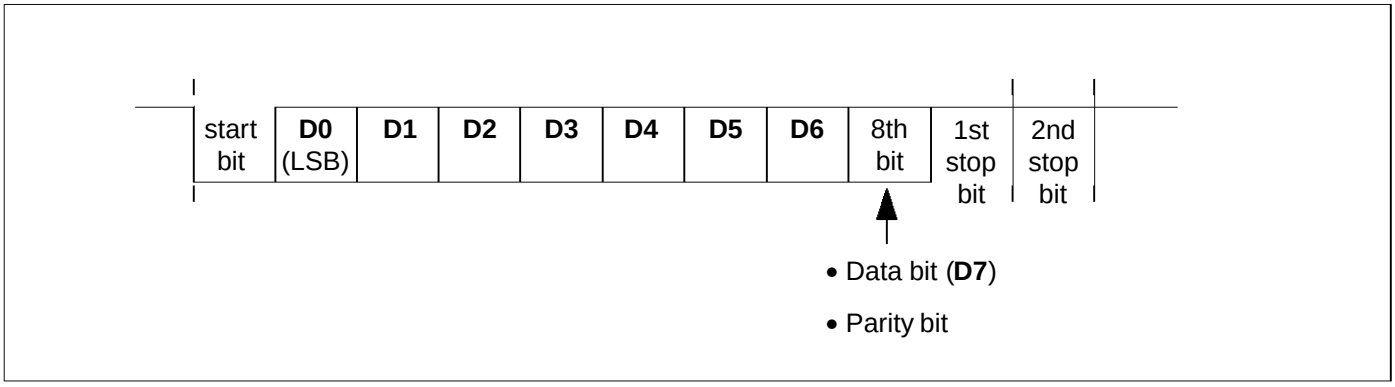


Figure 13.2 8-bit data frames

9-bit data frames

9-bit data frames consist of:

- nine data bits **D0-8**;
- eight data bits **D0-7** plus an automatically generated parity bit;
- eight data bits **D0-7** plus a wake-up bit.

Parity may be odd or even, depending on the **ParityOdd** bit in the **ASCControl** register. An even parity bit will be set, if the modulo-2-sum of the eight data bits is 1. An odd parity bit will be cleared in this case. The parity error flag (**ParityError**) will be set if a wrong parity bit is received. The parity bit itself will be stored in bit 8 of the **ASCRxBuffer** register, see Table 13.4.

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 can 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 + wake-up bit mode), so each slave can examine the 8 least significant bits (LSBs) of the received character (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 + wake-up bit mode, ignoring the following data bytes.

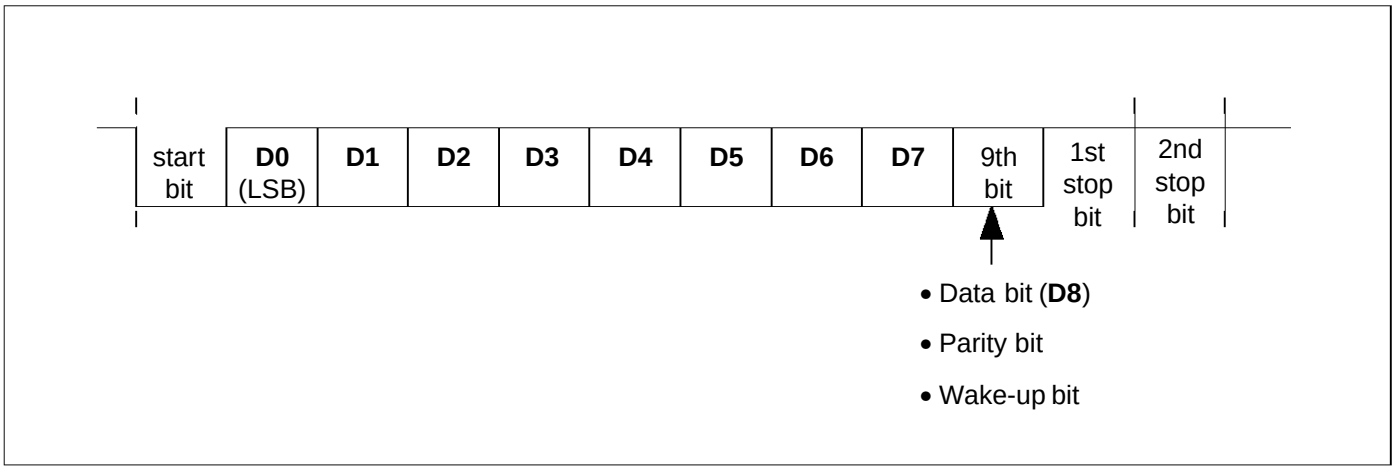


Figure 13.3 9-bit data frames

Transmission

Transmission begins at the next overflow of the divide-by-16 counter (see Figure 13.3 above), provided that the **Run** bit is set and data has been loaded into the **ASCTxBuffer**. 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, if selected)
- the stop bits (0.5, 1, 1.5 or 2 stop bits)

Data transmission is double buffered. When the transmitter is idle, the transmit data written into the transmit buffer is immediately moved to the transmit shift register, thus freeing the transmit buffer for the next data to be sent. This is indicated by the transmit buffer empty flag (**TxBufEmpty**) being set. The transmit buffer can be loaded with the next data, while transmission of the previous data is still going on.

The transmitter empty flag (**TxEmpty**) 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.

Reception

Reception is initiated by a falling edge on the data input pin (**RXD**), provided that the **Run** and **RxEnable** bits 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 is not a 0 when the start bit is sampled, the receive circuit is reset and waits for the next falling edge transition of the **RXD** pin. If the start bit is valid, 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.

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 (**ASCRxBuffer**). The receive buffer full flag (**RxBufFull**) is set, and the parity (**ParityError**) and framing error (**FrameError**) flags are updated, after the last stop bit has been received (at the end of the last stop bit programmed period), regardless of whether valid stop bits have been received or not. The receive circuit then waits for the next start bit (falling edge transition) at the **RXD** pin.

Reception is stopped by clearing the **RxEnable** bit. A currently received frame is completed including the generation of the receive status flags. Start bits that follow this frame will not be recognized.

Note: 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, the receive buffer full (**RxBufFull**) flag will not be set and no data will be transferred.

13.2 Hardware error detection capabilities

To improve the safety of serial data exchange, the ASC provides three error status flags in the **ASCStatus** register which indicate if an error has been detected during reception of the last data frame and associated stop bits.

The parity error (**ParityError**) bit is set when the parity check on the received data is incorrect.

The framing error (**FrameError**) bit 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 (**OverrunError**) bit is set when the last character received in the **ASCRxBuffer** 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.

13.3 Baud rate generation

The 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 **ASCControl** 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. The **ASCBaudRate** register is the dual-function baud rate generator/reload register. A read from this register returns the content of the timer, writing to it updates the reload register.

An auto-reload of the timer with the content of the reload register is performed each time the **ASCBaudRate** register is written to. However, if the **Run** bit is 0 at the time the write operation to the **ASCBaudRate** register is performed, the timer will not be reloaded until the first CPU clock cycle after the **Run** bit is 1.

13.3.1 Baud rates

The baud rate generator provides a clock at 16 times the baud rate. The baud rate and the required reload value for a given baud rate can be determined by the following formulas:

$$\text{Baud rate} = \frac{f_{CPU}}{16\ (<\text{ASCBaudRate}>)}$$

$$<\text{ASCBaudRate}> = \left(\frac{f_{CPU}}{16 \times \text{Baud rate}}\right)$$

where: <ASCBaudRate> represents the content of the **ASCBaudRate** register, taken as unsigned 16-bit integer,
f_{CPU} is the frequency of the CPU.

The table below lists various commonly used baud rates together with the required reload values and the deviation errors for an example baud rate with a CPU clock of 40 MHz. Note, this does not imply availability of a 40 MHz device.

Baud rate	Reload value (exact)	Reload value (integer)	Reload value (hex)	Deviation error
625 K	5	5	0005	0%
38.4 K	81.380	81	0051	0.5%
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	0.001%

Table 13.1 Baud rates

Note: The deviation errors given in the table above are rounded.

13.4 Interrupt control

The ASC contains two registers that are used to control interrupts, the status register (**ASCStatus**) and the interrupt enable register (**ASCIntEnable**). The status bits in the **ASCStatus** register deter-

mine the cause of the interrupt. Interrupts will occur when a status bit is 1 (high) and the corresponding bit in the **ASCIntEnable** register is 1.

The error interrupt signal (**ErrorInterrupt**) 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 **ASCIntEnable** register.

An overall interrupt request signal (**ASC_interrupt**) is generated from the OR of the **ErrorInterrupt** signal and the **TxEmpty**, **TxBufEmpty** and **RxBufFull** signals.

Note: the status register *cannot* be written directly by software. 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.

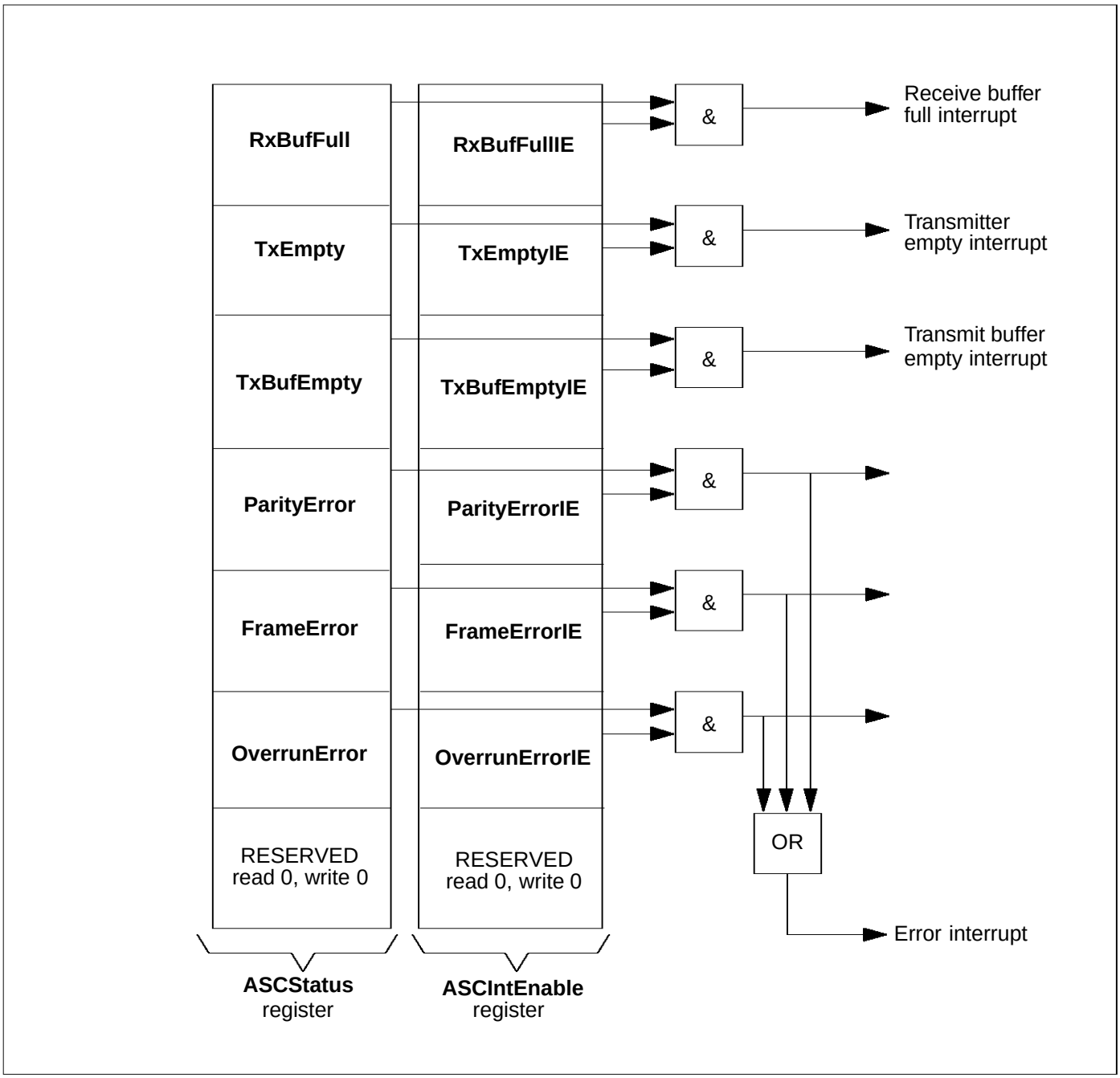


Figure 13.4 ASC status and interrupt registers

13.4.1 Using the ASC interrupts

For normal operation (i.e. besides the error interrupt) the ASC provides three interrupt requests to control data exchange via the serial channel:

- **TxBufEmpty** is activated when data is moved from **ASCTxBuffer** 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 **ASCRxBuffer**.

The transmitter generates two interrupts. This provides advantages for the servicing software.

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. This leaves just one bit-time for the handler to respond to the transmitter interrupt request.

Using the transmit buffer interrupt (**TxBufEmpty**) to reload transmit data allows the time to transmit a complete frame for the service routine, as **ASCTxBuffer** may be reloaded while the previous data is still being transmitted.

As shown in Figure 13.5 below, **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.

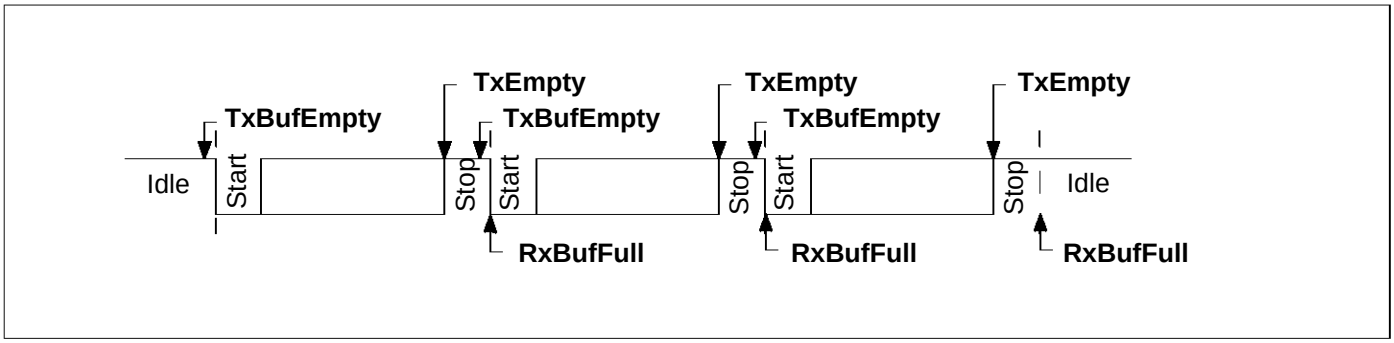


Figure 13.5 ASC interrupt generation

13.5 ASC configuration registers

ASCBaudRate register

The **ASCBaudRate** register is the dual-function baud rate generator/reload register.

A read from this register returns the content of the timer, writing to it updates the reload register.

An auto-reload of the timer with the content of the reload register is performed each time the **ASCBaudRate** register is written to. However, if the **Run** bit of the **ASCControl** register, see Table 13.5, is 0 at the time the write operation to the **ASCBaudRate** register is performed, the timer will not be reloaded until the first CPU clock cycle after the **Run** bit is 1.

ASCBaudRate		ASC base address + #00Read/Write	
Bit	**Bit field**	**Write Function**	**Read Function**
15:0	**ReloadVal**	16-bit reload value	16-bit count value

Table 13.2 **ASCBaudRate** register format

ASCTxBuffer register

Writing to the transmit buffer register starts data transmission.

ASCTxBuffer		ASC base address + #04Write 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 in 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 in the **ASCControl** register). Note: If the **Mode** field selects an 8-bit frame then this bit should be written as 0.
15:9		RESERVED. Write 0.

Table 13.3 **ASCTxBuffer** register format

ASCRxBuffer register

The received data and, if provided by the selected operating mode, the received parity bit can be read from the receive buffer register.

ASCRxBuffer		ASC base address + #08Read 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 in the **ASCControl** register). Note: If the **Mode** field selects a 7-bit frame then this bit is undefined. Software should ignore this bit when reading 7-bit frames.
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 in the **ASCControl** register). Note: If the **Mode** field selects a 7- or 8-bit frame then this bit is undefined. Software should ignore this bit when reading 7- or 8-bit frames.
15:9		RESERVED. Will read back 0.

Table 13.4 **ASCRxBuffer** register format

ASCControl register

This register controls the operating mode of the ASC and contains control bits for mode and error check selection, and status flags for error identification.

Note: Programming the mode control field (**Mode**) to one of the reserved combinations may result in unpredictable behavior.

Note: 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.

ASCControl		ASC base address + #0CRead/Write
Bit	**Bit field**	**Function**
2:0	**Mode**	ASC mode control **Mode2:0** **Mode** 000 RESERVED 001 8-bit data 010 RESERVED 011 7-bit data + parity 100 9-bit data 101 8-bit data + wake up bit 110 RESERVED 111 8-bit data + parity
4:3	**StopBits**	Number of stop bits selection **StopBits1:0** **Number of stop bits** 00 0.5 stop bits 01 1 stop bit 10 1.5 stop bits 11 2 stop bits
5	**ParityOdd**	Parity selection 0 Even parity (parity bit set on odd number of ‘1’s in data) 1 Odd parity (parity bit set on even number of ‘1’s in data)
6	**LoopBack**	Loopback mode enable bit 0 Standard transmit/receive mode 1 Loopback mode enabled
7	**Run**	Baud rate generator run bit 0 Baud rate generator disabled (ASC inactive) 1 Baud rate generator enabled
8	**RxEnable**	Receiver enable bit 0 Receiver disabled 1 Receiver enabled
9	**SCEnable**	SmartCard enable bit 0 SmartCard mode disabled 1 SmartCard mode enabled
15:10		RESERVED. Write 0, will read back 0.

Table 13.5 **ASCControl** register format

ASCIntEnable register

The **ASCIntEnable** register enables a source of interrupt.

Interrupts will occur when a status bit in the **ASCStatus** register is 1, and the corresponding bit in the **ASCIntEnable** register is 1.

ASCIntEnable	**ASC base address + #10Read/Write**	
Bit	**Bit field**	**Function**
0	**RxBufFullIE**	Receiver buffer full interrupt enable 0 receiver buffer full interrupt disable 1 receiver buffer full interrupt enable
1	**TxEmptyIE**	Transmitter empty interrupt enable 0 transmitter empty interrupt disable 1 transmitter empty interrupt enable
2	**TxBufEmptyIE**	Transmitter buffer empty interrupt enable 0 transmitter buffer empty interrupt disable 1 transmitter buffer empty interrupt enable
3	**ParityErrorIE**	Parity error interrupt enable 0 parity error interrupt disable 1 parity error interrupt enable
4	**FrameErrorIE**	Framing error interrupt enable 0 framing error interrupt disable 1 framing error interrupt enable
5	**OverrunErrorIE**	Overrun error interrupt enable 0 overrun error interrupt disable 1 overrun error interrupt enable
7:6		RESERVED. Write 0, will read back 0.

Table 13.6 **ASCIntEnable** register format

ASCStatus register

The **ASCStatus** register determines the cause of an interrupt.

ASCStatus	**ASC base address + #14Read Only**	
Bit	**Bit field**	**Function**
0	**RxBufFull**	Receiver buffer full flag 0 receiver buffer not full 1 receiver buffer full
1	**TxEmpty**	Transmitter empty flag 0 transmitter not empty 1 transmitter empty
2	**TxBufEmpty**	Transmitter buffer empty flag 0 transmitter buffer not empty 1 transmitter buffer empty
3	**ParityError**	Parity error flag 0 no parity error 1 parity error
4	**FrameError**	Framing error flag 0 no framing error 1 framing error
5	**OverrunError**	Overrun error flag 0 no overrun error 1 overrun error
7:6		RESERVED. Write 0, will read back 0.

Table 13.7 **ASCStatus** register format

13.6 SmartCard mode specific operation

The **ASCGuardTime** register enables the user to define a programmable number of baud clocks to delay the assertion of **TxEmpty**.

ASCGuardTime		**ASC base address + #18Read/Write**
Bit	**Bit field**	**Function**
7:0	**GuardTime**	Number of baud clocks to delay assertion of **TxEmpty**.
15:8		RESERVED. Write 0, will read back 0.

Table 13.8 **ASCGuardTime** register

To conform to the ISO Smart Card 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.

  Note: The loopback in smartcard mode feeds back an undelayed signal from the transmitter shift register to the receive shift register. The signal at the **TXD** pin is still delayed.

- If a parity error is detected during transmission 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** flag can be delayed by programming the **ASCGuardTime** register. 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 **ASCGuardTime** register. **TxEmpty** is held low during this time. When the guardtime counter reaches the programmed value **TxEmpty** is asserted high.

  The de-assertion of **TxEmpty** is unaffected by SmartCard mode.

- An additional ASC output enable pin (**notOE**) is available which can be used to control an external driver. The **notOE** signal is low during the start, data and parity bits of a frame and high at all other times in normal mode. In SmartCard mode it also mimics the behavior of the **TXD** pin when a parity error is detected.

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.

14 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.

A UART (ASC) configured as eight data bits plus parity, 0.5 or 1.5 stop bits, with SmartCard mode enabled provides the UART function of the SmartCard interface. 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*, inverted data and MSB first, is handled in software.

Refer to “UART interface (ASC)” on page 84 and “Parallel Input/Output” on page 118 for details of the ASC and PIO ports.

14.1 External interface

The signals required by the SmartCard are given in Table 14.1.

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 14.1 SmartCard pins

The signals provided on the ST20-TP1 are given in Table 14.2.

Pin	In/Out	Function
ScClk	out, open drain for 5 V 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 detect.

Table 14.2 SmartCard interface pins

The **ScRST**, **ScCmdVpp**, **ScCmdVcc**, and **ScDetect** signals are provided by PIO bits of the PIO ports. Programming the PIO bits of the port for alternate 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 bit assignments can be found in Table 29.1 on page 147.

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 input to 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 ST20 device to the SmartCard, the ASC will need to be set up in SmartCard mode.

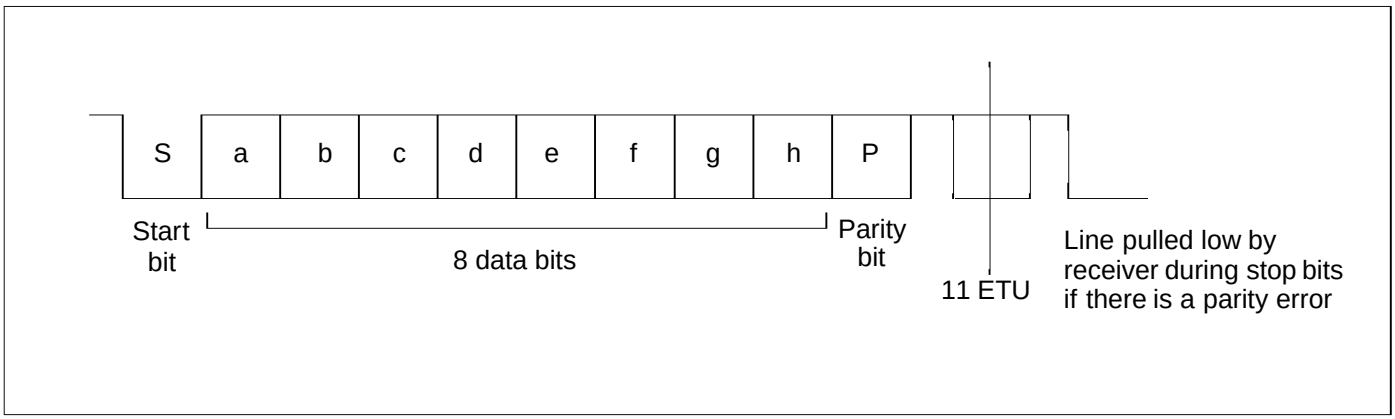


Figure 14.1 ISO 7816-3 asynchronous protocol

14.2 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, therefore updates to the clock rate must be synchronized to the clock (**Clk**) to the SmartCard, i.e. 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 or an external pin. Two registers control the period of the clock and the running of the clock.

Note: The clock generator is independent of the UART Baud rate.

14.2.1 SmartCard clock generator registers

The SmartCard can be programmed via registers which are mapped into the device address space. They may be accessed using *devsw* and *devlw* instructions.

The base addresses for the SmartCard registers are given in the Memory Map chapter.

Note: During reset all of the registers are reset to ‘0’.

ScClkVal register

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.

ScClkVal	SmartCard clock generator base address + #00	Write only
Bit	**Bit field**	**Function**
4:0	**ScClkVal**	These bits determine the source clock divider value. This value multiplied by 2 gives the clock division factor, see examples which follow: **ScClkVal4:0** **Division** 00000 DO NOT PROGRAM THIS VALUE 00001 divides the source clock frequency by 2 00010 divides the source clock frequency by 4
7:5		RESERVED. Write 0.

Table 14.3 **ScClkVal** register format

ScClkCon register

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.

ScClkCon	SmartCard clock generator base address + #04	Write only
Bit	**Bit field**	**Function**
0	**ScClkSource**	Selects source of SmartCard clock. 0 selects global clock 1 selects external pin
1	**ScClkEnable**	SmartCard clock generator enable bit. 0 stop clock, set output low and reset divider 1 enable clock generator
7:2		RESERVED. Write 0.

Table 14.4 **ScClkCon** register format

15 I^2C 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 via an I^2C bus or other serial bus standard.

The SSC can be used to communicate with shift registers (IO expansion), peripherals (e.g. EEPROMs) or other controllers (networking).

Various interface standards exist for these, the most important of which is the I^2C bus in the set-top box application as this is the interface used most often for the control of the Link-IC and the PAL/NTSC encoder. Figure 15.1 below shows how the SSC is interfaced to an I^2C bus as the bus master. Software is required to handle some of the I^2C bus protocol such as byte acknowledgement.

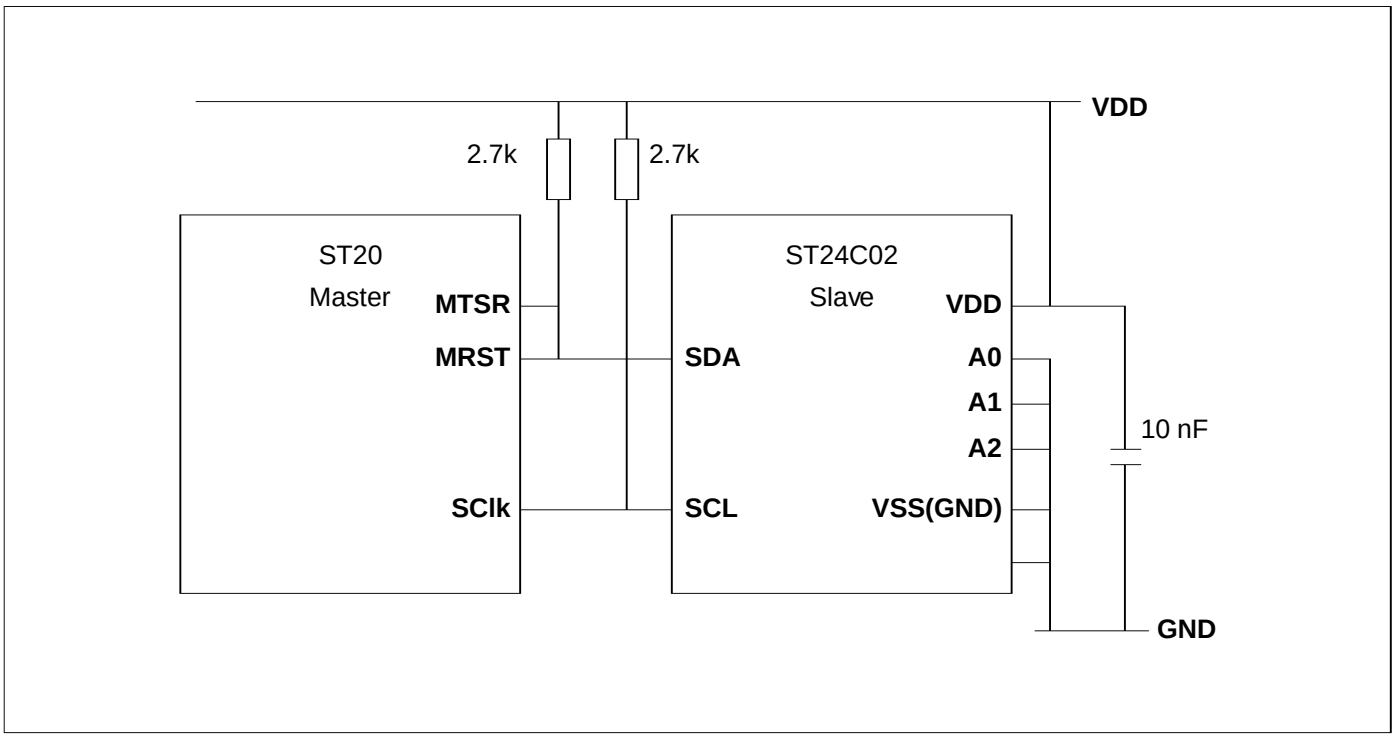


Figure 15.1 Connection of ST24C02 and ST20 in I^2C-bus

The SSC supports full-duplex and 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.

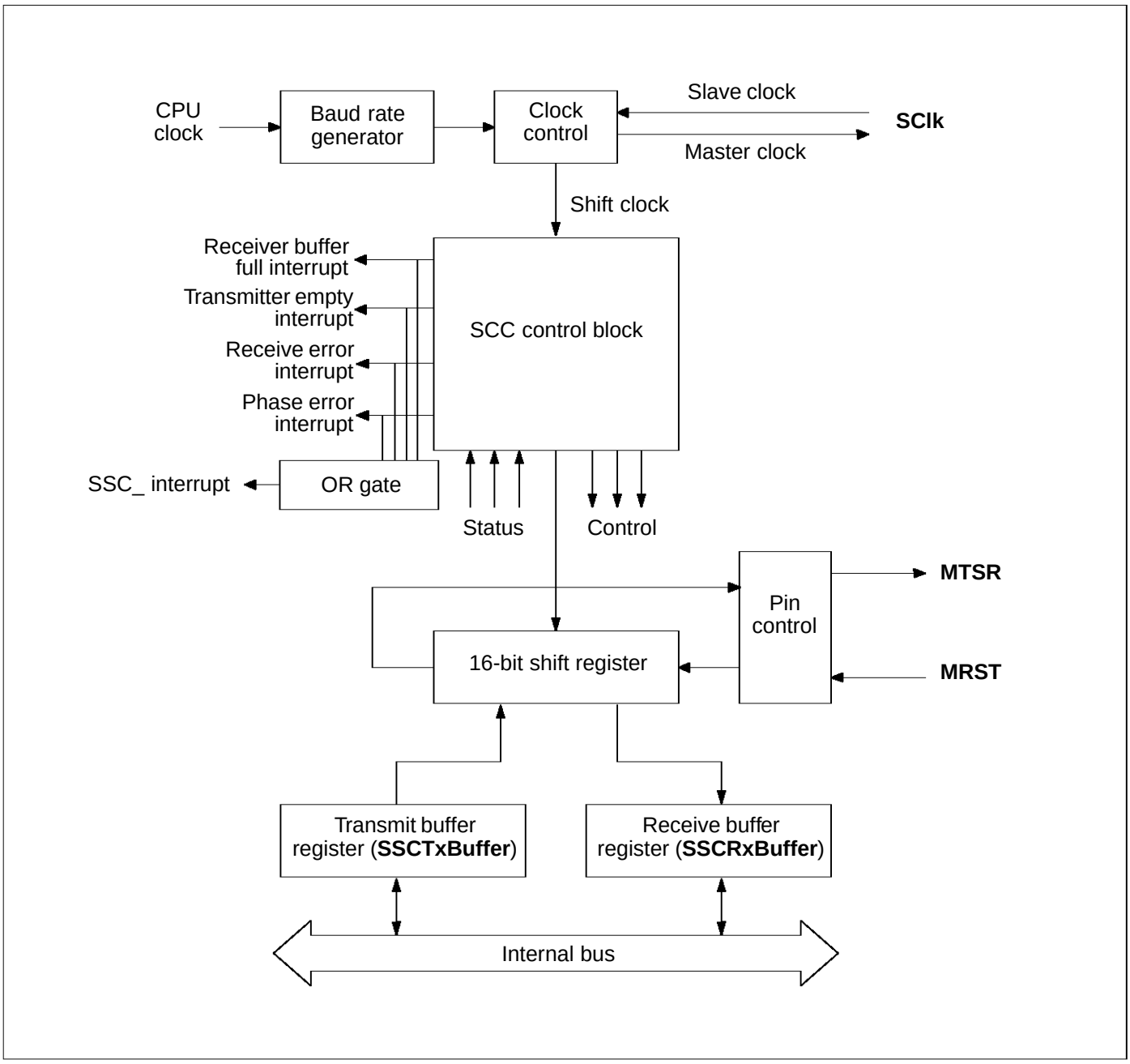


Figure 15.2 Block diagram of the SSC

15.1 Synchronous serial controller operation

The operating mode of the serial channel SSC is controlled by the control register (**SSCControl**), see Table 15.5.

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 15.2). 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 reg-

ister 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.

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.

15.1.1 Clock control

If the **ClkPhase** and **ClkPolarity** bits in the **SSCControl** register are programmed, as defined by Table 15.5 on page 111, then the clock and data relationship will be I^2C compatible. The high level of the clock is stable during the data and I^2C setup and hold times are met.

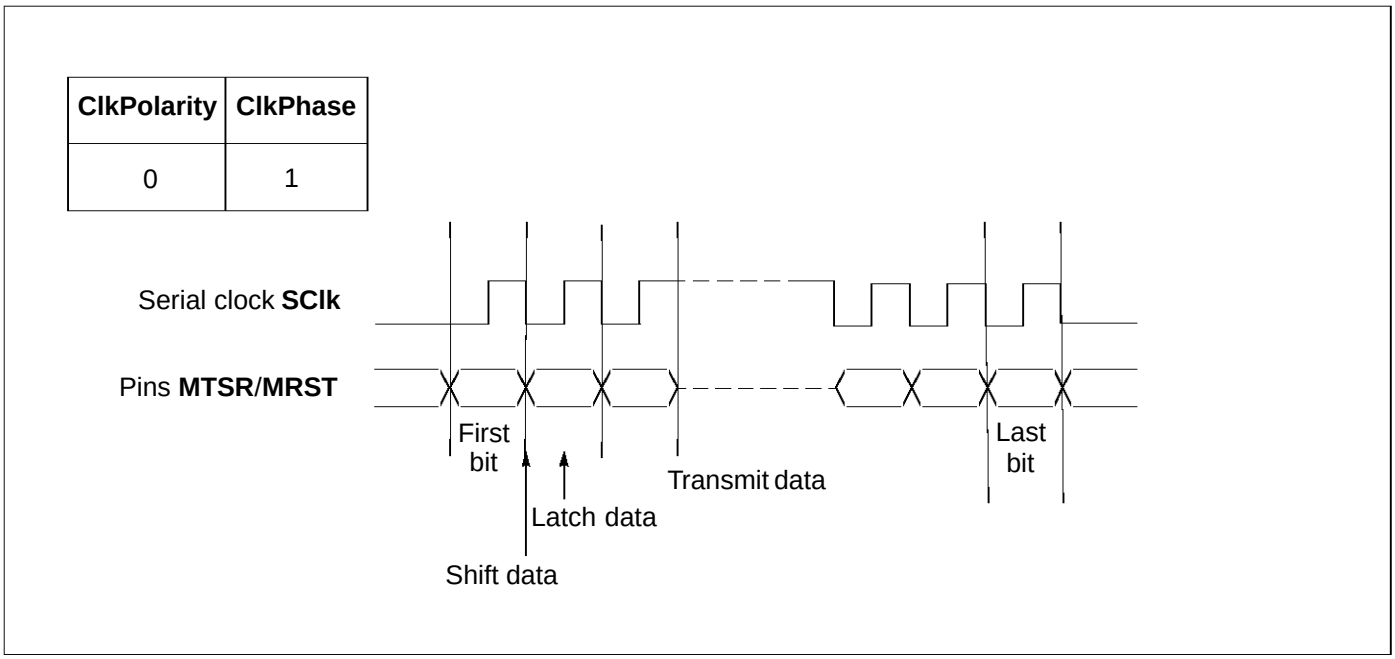


Figure 15.3 Clock and data relationships

15.1.2 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.

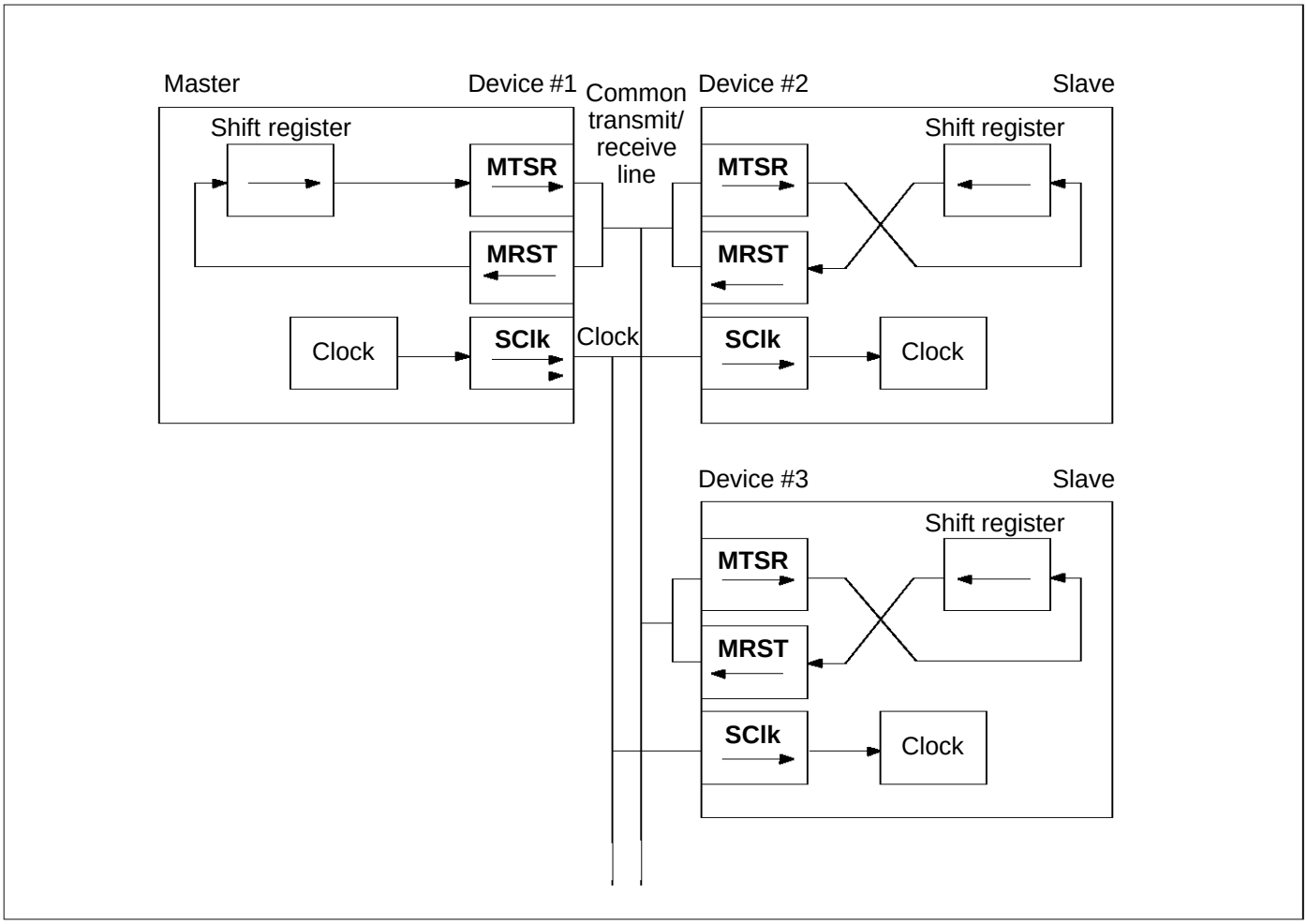


Figure 15.4 Half-duplex configuration

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, **MTSR** for a slave). 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.

15.2 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 **SSCStatus** 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, by the **RxErrorIE** bit in the **SSCIntEnable** register, 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 Section 15.1.1 on page 105). This condition sets the error flag (**PhaseError**) and when enabled, by the **PhaseErrorIE** bit in the **SSCIntEnable** register, the error interrupt request flag (**ErrorInterrupt**).

15.3 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.

15.3.1 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:

$$\text{Baudrate} = \frac{f_{CPU}}{2 \times \langle \text{SSCBaudRate} \rangle} \qquad \langle \text{SSCBaudRate} \rangle = \left(\frac{f_{CPU}}{2 \times \text{Baudrate}}\right)$$

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 15.1 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 15.1 Baud rates and bit times for different **SSCBaudRate** reload values

Note: The content of **SSCBaudRate** must be > 0.

15.4 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 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**).

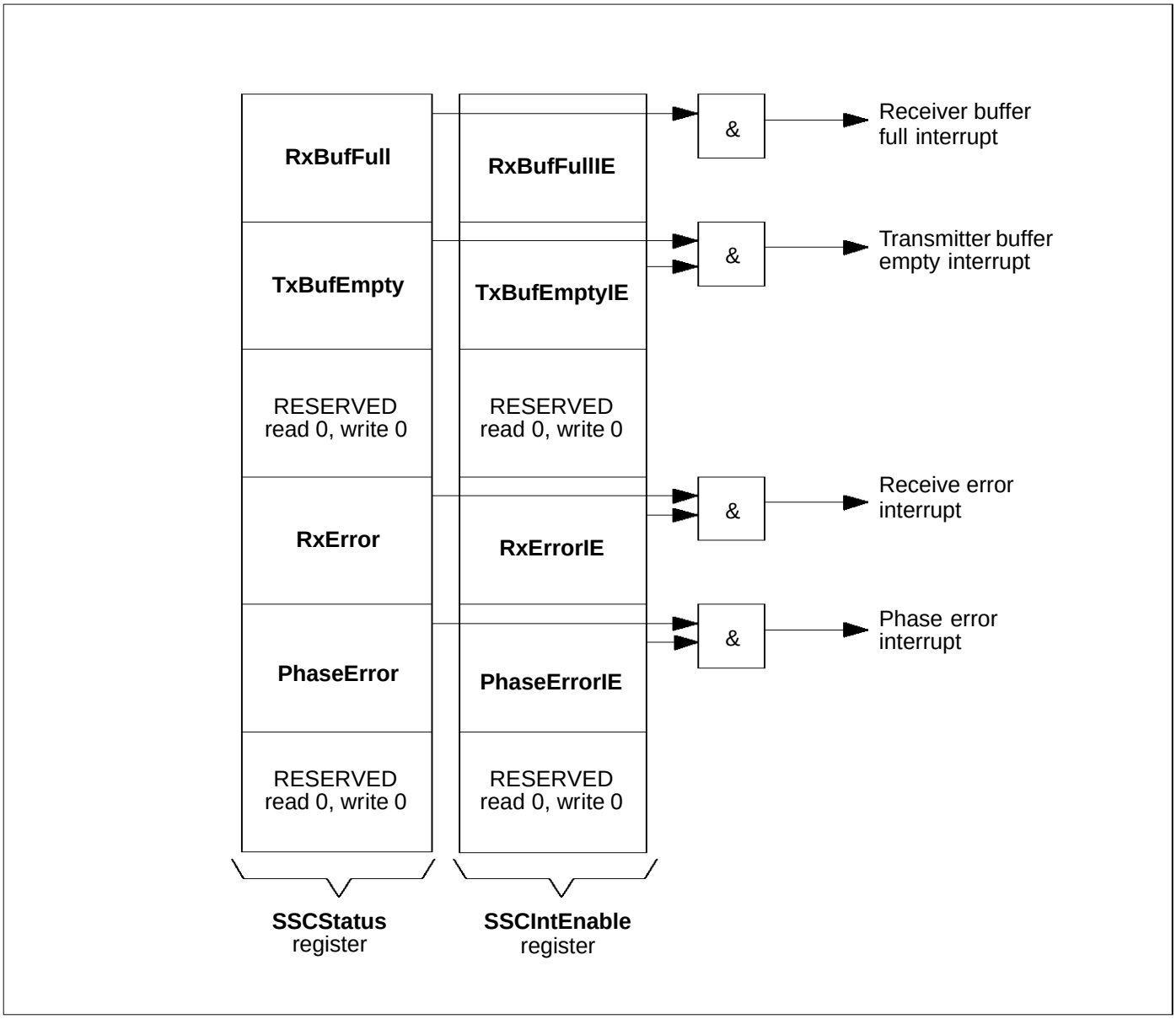


Figure 15.5 SSC status and interrupt registers

15.4.1 Using the SSC interrupts

An interrupt handler for the SSC needs to read the **SSCStatus** register before writing the **SSCTxBuffer** or reading the **SSCRxBuffer** 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.

15.5 SSC configuration registers

SSCBaudRate register

The **SSCBaudRate** register is the dual function 16-bit baud rate generator/reload register. Note that the resultant baud rate for transmission and reception is *half* the value in the **SSCBaudRate** register.

Note: the content of the **SSCBaudRate** register must be > 0.

SSCBaudRate		**ASC base address + #00Read/Write**	
Bit	**Bit field**	**Write function**	**Read function**
15:0	**ReloadVal**	16-bit reload value	16-bit count value

Table 15.2 **SSCBaudRate** register format

SSCTxBuffer register

Transmit data is written into the transmit buffer register.

Any unused bits of the **SSCTxBuffer** register are ignored.

SSCTxBuffer		**SSC base address + #04Write only**
Bit	**Bit field**	**Function**
15:0	**TD15:0**	Transmit buffer data **D15:0**

Table 15.3 **SSCTxBuffer** register format

SSCRxBuffer register

The received data can be read from the receive buffer register.

Any unused bits of the **SSCRxBuffer** register are invalid and should be ignored by the receiver service routine.

SSCRxBuffer		**SSC base address + #08Read only**
Bit	**Bit field**	**Function**
15:0	**RD15:0:**	Receive buffer data **D15:0**

Table 15.4 **SSCRxBuffer** register format

SSCControl register

The operating mode of the SSC is controlled by the control register.

SSCControl		**SSC base address +#0CRead/Write**
Bit	**Bit field**	**Function**
3:0	**DataWidth**	Data width selection **DataWidth3:0 Data width** 0000 RESERVED. Do not use this combination. 0001 2 bits 0010 3 bits ... ... 1111 16 bits
4	**HeadControl**	Heading control bit For I^2C operation, software *must* write a 1; the effect of writing 0 is undefined. The most significant bit (MSB) of the selected data width is shifted out first.
5	**ClkPhase**	Clock phase control bit For I^2C operation, software *must* write a 1; the effect of writing 0 is undefined.
6	**ClkPolarity**	Clock polarity control bit For I^2C operation, software *must* write a 0; the effect of writing 1 is undefined.
8	**MasterSel**	Master select bit For I^2C operation, software *must* write a 1; the effect of writing 0 is undefined.
9	**Enable**	Enable bit 0 transmission and reception disabled 1 transmission and reception enabled
10	**LoopBack**	Loopback bit 0 transmitter is connected to shift register input 1 shift register output is connected to shift register input
7, 15:11		RESERVED. Write 0, read back 0.

Table 15.5 **SSCControl** register format

SSCIntEnable register

The **SSCIntEnable** register enables a source of interrupt.

Interrupts will occur when a status bit in the **SSCStatus** register is 1, and the corresponding bit in the **SSCIntEnable** register is 1.

SSCIntEnable		SSC base address + #10Read/Write
Bit	**Bit field**	**Function**
0	**RxBufFullIE**	Receiver buffer full interrupt enable 0 receiver buffer full interrupt disable 1 receiver buffer full interrupt enable
1	**TxBufEmptyIE**	Transmitter buffer empty interrupt enable 0 transmitter buffer empty interrupt disable 1 transmitter buffer empty interrupt enable
3	**RxErrorIE**	Receive error interrupt enable 0 receive error interrupt disable 1 receive error interrupt enable
4	**PhaseErrorIE**	Phase error interrupt enable 0 phase error interrupt disable 1 phase error interrupt enable
2, 7:5		RESERVED. Write 0, will read back 0.

Table 15.6 **SSCIntEnable** register format

SSCStatus register

The **SSCStatus** register determines the cause of an interrupt.

SSCStatus		SSC base address + #14Read Only
Bit	**Bit field**	**Function**
0	**RxBufFull**	Receiver buffer full flag 0 receiver buffer not full 1 receiver buffer full
1	**TxBufEmpty**	Transmitter buffer empty flag 0 transmitter buffer not empty 1 transmitter buffer empty
3	**RxError**	Receive error flag 0 no receive error 1 receive error set
4	**PhaseError**	Phase error flag 0 no phase error 1 phase error set
2, 7:5		RESERVED. Will read back 0.

Table 15.7 **SSCStatus** register format

16 PWM and counter module

This module includes two separate 8-bit counters used for pulse width modulation (PWM) and two 32-bit counters with capture registers. The counters can be clocked from a pre-scaled internal clock or from a pre-scaled external clock via the **CaptureClk** input. The event on which the timer value is captured is also programmable.

The PWM and counter module generates a single interrupt signal, the exact event causing the interrupt can be determined from the **CaptureStatus** register. The interrupts are cleared by writing a 1 to the corresponding bits in the **CaptureAck** register.

16.1 External interface

Pin	In/Out	Function
PWMOut0-1	out	PWM outputs
CaptureIn0-1	in	Capture trigger inputs
CaptureClk0-1	in	External capture counter clocks

Table 16.1 PWM and counter pins

16.2 PWM and counter control registers

The PWM and counter module is programmable via control registers.

The base address for the PWM control registers are given in the Memory map.

PWMVal0-1 registers

The **PWMVal0-1** registers contain the counter value for each of the 8-bit PWM counters.

PWMVal0-1	**PWM base address + #00 to #04**	**Read/Write**
Bit	**Bit field**	**Function**
7:0	**PWMVal**	8-bit PWM counter value, see Figure 16.1.

Table 16.2 **PWMVal0-1** registers format

This value is used to determine the width of the pulse generated on the **PWMOut** pin, see Figure 16.1.

PWMOut pulse width = (**PWMVal** + 1) X prescaled clock period

If **PWMVal** = 0, **PWMOut** pulse width = 1 prescaled clock cycle.

If **PWMVal** = 255, **PWMOut** pulse width = 256 prescaled clock cycles, i.e. **PWMOut** does not go low.

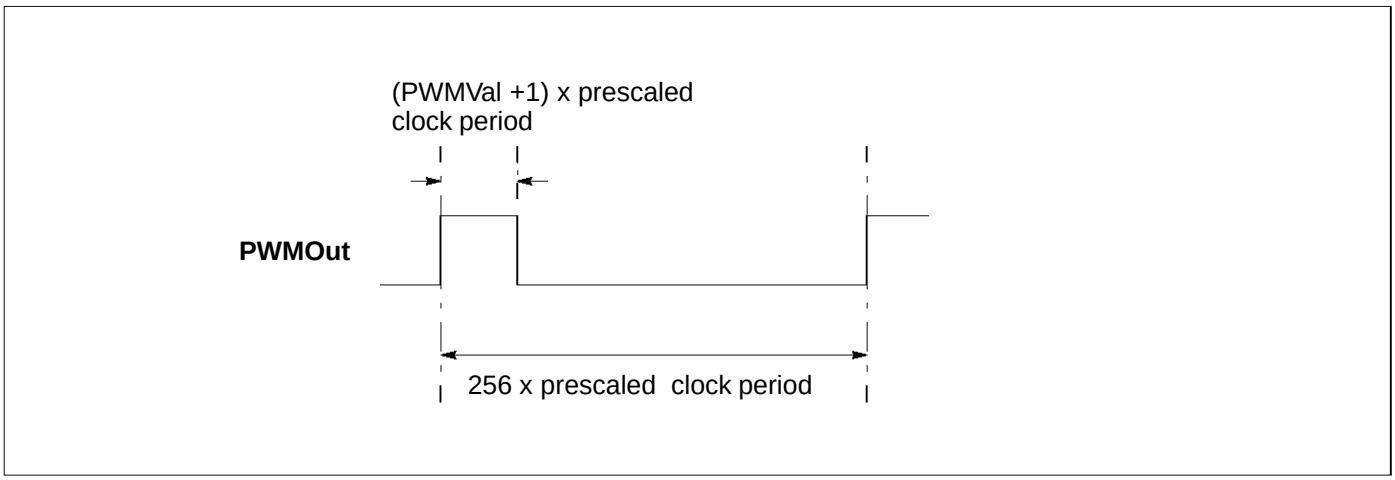


Figure 16.1 PWM counter value

The clock used in this module, either **ClockIn** or **CaptureClk**, is selected by the **PWMClkSource** bit of the **CaptureControl** register. This clock can be further prescaled by programming the **PWM-ClkVal** bit field. The prescaler divides the selected clock by **PWMClkVal**+1.

The PWM counter is enabled by setting the **PWMEnable** bit of the **CaptureControl** register. When it is disabled (**PWMEnable** is 0), **PWMOut** is forced low.

When the PWM counter overflows an interrupt is generated if the **PWMInterrupt** bit is set.

CaptureVal0-1 registers

The **CaptureVal0-1** registers contain the captured value of each of the 32-bit capture counters.

CaptureVal0-1	**PWM base address + #08 to #0C**	**Read only**
Bit	**Bit field**	**Function**
31:0	**CaptureVal**	32-bit capture counter value

Table 16.3 **CaptureVal0-1** registers format

The clock used in this module, either **ClockIn** or **CaptureClk**, is selected by the **CaptureClk-Source** bit of the **CaptureControl** register. This clock can be further prescaled by programming the **CaptureClkVal** bit field. The prescaler divides the selected clock by **CaptureClkVal** + 1.

The event which causes the capture of the counter value is selected by the **CaptureEvent** bit to be either **CaptureIn** or **LPacketClk**. It can be set to capture on a rising or falling edge determined by the setting of the **CaptureEdge** bit.

An interrupt is generated when a capture event occurs if the **CaptureInterrupt** bit is set.

The counter is enabled by setting the **CaptureEnable** bit. Any capture events which occur when the counter is disabled will be ignored, with neither the counter value being captured nor an interrupt being generated.

CaptureControl register

The **CaptureControl** register is used to set the pre-scalers and clock sources for the PWM and capture counters, to control the interrupts, and to configure the capture signal source for the capture registers.

CaptureControl	PWM base address + #10Read/Write	
Bit	**Bit field**	**Function**
0	**PWM0Interrupt**	PWM0 interrupt enable 1 interrupt on 8-bit counter overflow
1	**PWM1Interrupt**	PWM1 interrupt enable 1 interrupt on 8-bit counter overflow
2	**Capture0Interrupt**	Capture0 interrupt enable 1 interrupt on capture event
3	**Capture1Interrupt**	Capture1 interrupt enable 1 interrupt on capture event
4	**PWM0ClkSource**	PWM0 clock source 0 ClockIn 1 CaptureClk0
8:5	**PWM0ClkVal**	PWM0 clock prescale value. The selected clock (**ClockIn** or **CaptureClk0**) is divided by **PWM0ClkVal** +1, for example: **PWM0ClkVal8:5** prescale value 0000 divide selected clock by 1 0100 divide selected clock by 5
9	**PWM1ClkSource**	PWM1 clock source 0 ClockIn 1 CaptureClk1
13:10	**PWM1ClkVal**	PWM1 clock prescale value. The selected clock is divided by **PWM1ClkVal** +1.
14	**Capture0ClkSource**	Capture0 clock source 0 ClockIn 1 CaptureClk0
18:15	**Capture0ClkVal**	Capture0 clock prescale value. The selected clock is divided by **Capture0ClkVal**+1.
19	**Capture0Event**	Capture0 capture event source 0 CaptureIn0 1 LPacketClk
20	**Capture0Edge**	Capture0 capture edge 0 rising edge 1 falling edge
21	**Capture1ClkSource**	Capture1 clock source 0 ClockIn 1 CaptureClock1

Table 16.4 **CaptureControl** register format

CaptureControl	PWM base address + #10	Read/Write
Bit	**Bit field**	**Function**
25:22	**Capture1ClkVal**	Capture1 clock prescale value. The selected clock is divided by **Capture1ClkVal**+1.
26	**Capture1Event**	Capture1 capture event source 0 CaptureIn1 1 LPacketClk
27	**Capture1Edge**	Capture1 capture edge 0 rising edge 1 falling edge
28	**PWM0Enable**	PWM0 enable 1 enables PWM0
29	**PWM1Enable**	PWM1 enable 1 enables PWM1
30	**Capture0Enable**	Capture0 enable 1 enables Capture0
31	**Capture1Enable**	Capture1 enable 1 enables Capture1

Table 16.4 **CaptureControl** register format

CaptureStatus register

This register is read only and determines the event which caused the interrupt.

An overall interrupt signal is generated from the OR of these 4 interrupts.

CaptureStatus	PWM base address + #14	Read only
Bit	**Bit field**	**Function**
0	**PWM0Int**	PWM0 interrupt 1 interrupt
1	**PWM1Int**	PWM1 interrupt 1 interrupt
2	**Capture0Int**	Capture0 interrupt 1 interrupt
3	**Capture1Int**	Capture1 interrupt 1 interrupt
7:4		RESERVED. Will read back 0.

Table 16.5 **CaptureStatus** register format

CaptureAck register

This register is write only. When a bit is set to 1 it clears the associated interrupt.

CaptureAck	**PWM base address + #18**	**Write only**
Bit	**Bit field**	**Function**
0	**PWM0IntAck**	PWM0 interrupt acknowledge 1 clears interrupt
1	**PWM1IntAck**	PWM1 interrupt acknowledge 1 clears interrupt
2	**Capture0IntAck**	Capture0 interrupt acknowledge 1 clears interrupt
3	**Capture1IntAck**	Capture1 interrupt acknowledge 1 clears interrupt
7:4		RESERVED. Write 0.

Table 16.6 **CaptureAck** register format

17 Parallel Input/Output

The ST20-TP1 device has 32 bits of Parallel Input/Output (PIO), configured in groups of eight bits, each bit being programmable as an output, an input, a bidirectional pin, or as an alternate function output pin. The alternate function connects signals from device peripherals to the pins of the device through the PIO.

Details of the alternate function assignments can be found in the Device Configuration chapter.

Each group of eight input bits can also be compared against a register and an interrupt generated when the value is not equal.

Output drivers for the PIO pins, both in PIO mode and the alternate function mode, can be programmed to be push-pull, open drain, or weak pull-up. The weak pull-up configuration avoids the need for pull-up resistors on unused pins while still allowing them to be driven for test purposes.

Each of the groups of eight bits operates as described in the following section.

17.1 PIO Ports0-3

Each of the eight bits of a PIO port has a corresponding bit in the PIO registers associated with each port. These registers hold: output data for the port (**PxOut**); the input data read from the pin (**PxIn**); PIO bit configuration registers (**PxC0**, **PxC1** and **PxC2**); and the two input compare function registers (**PxComp** and **PxMask**).

All of the registers, except the **PxIn** registers, are each mapped onto three separate 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.

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.

Note that during reset all the registers are reset to ‘00000000’.

17.1.1 PIO Data registers

PxOut0-3 register

This register holds output data for the port.

The base addresses for the PIOx registers are given in the Memory Map.

PxOut0-3	**PIOx base address + #00**	**Read/Write**
Bit	**Bit field**	**Function**
7:0	**PxOut7:0**	Bits 0 to 7 of output data for the port.

Table 17.1 **PxOut0-3** register format — 1 register per port

PxIn0-3 register

The data read from this register will give the logic level present on an input pin of the port at the start of the read cycle to this register. The read data will be the last value written to the register regardless of the pin configuration selected.

PxIn0-3	**PIOx base address + #10**	**Read only**
Bit	**Bit field**	**Function**
7:0	**PxIn7:0**	Bits 0 to 7 of input data for the port.

Table 17.2 **PxIn** register format — 1 register per port

17.1.2 PIO Configuration registers

There are three configuration registers (**PxC0**, **PxC1** and **PxC2**) which are used to configure each of the PIO port bits as an input, output, bidirectional, or alternate function pin (if any), with options for the output driver configuration.

PxC0-2	**PIOx base address + #20 to #40**	**Read/Write**
Bit	**Bit field**	**Function**
7:0	**ConfigData7:0**	PIO Configuration data bits 0 to 7.

Table 17.3 **PxC0-2** registers format — 3 registers per port

The selections made by the bits in these registers for each I/O bit are given in Table 17.4 below.

PxBitn configuration	**PxBitn output**	**PxC2n**	**PxC1n**	**PxC0n**
Weak Pull-up	Weak Pull-up	0	0	0
Bidirectional	Open drain	0	0	1
Output	Push-pull	0	1	0
Bidirectional	Open drain	0	1	1
Input	Hi-Z	1	0	0
Input	Hi-Z	1	0	1
Alternate function output	Push-pull	1	1	0
Alternate function bidirectional	Open drain	1	1	1

Table 17.4 PIO port bits configurations

17.1.3 PIO Input compare and Compare mask registers

The Input compare register (**PxComp**) holds the value to which the input data from the PIO ports pins will be compared. If any of the input bits are different from the corresponding bits in the **PxComp** register and the corresponding bit position in the PIO Compare mask register (**PxMask**) is set to 1, then the internal interrupt signal for the port will be set to 1.

The compare function is sensitive to changes in levels on the pins and so the change in state on the input pin must be greater in duration than the interrupt response time for the compare to be seen as a valid interrupt by an interrupt service routine.

Note that the compare function is operational in all configurations for a PIO bit including the alternate function modes.

PxComp	**PIOx base address + #50**	**Read/Write**
Bit	**Bit field**	**Function**
7:0	**PxComp7:0**	Bit 0 to 7 value to which the input data from the PIO port pins will be compared.

Table 17.5 **PxComp** register format — 1 register per port

PxMask	**PIOx base address + #60**	**Read/Write**
Bit	**Bit field**	**Function**
7:0	**PxMask7:0**	When set to 1, the compare function for the internal interrupt for the port is enabled. If the respective bit (0 to 7) of the input is different to the respective **PxComp7:0** bit in the **PxComp** register, then an interrupt is generated.

Table 17.6 **PxMask** register format — 1 register per port

18 Serial link interface (OS-Link)

The ST20-TP1 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 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.

18.1 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 18.1). 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.

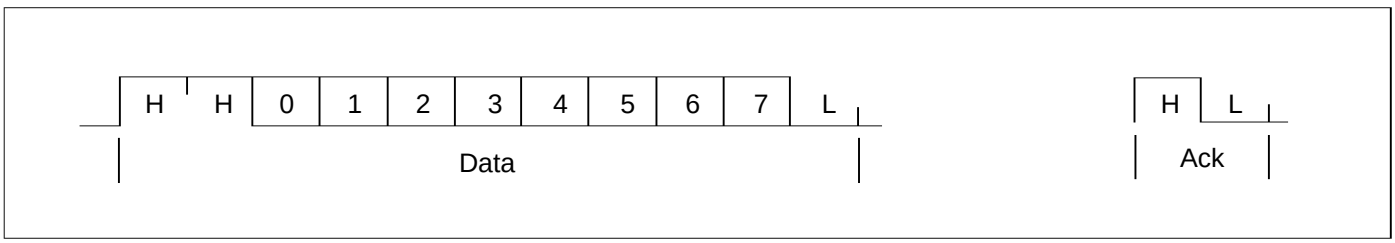


Figure 18.1 OS-Link data and acknowledge formats

18.2 OS-Link speed

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

18.3 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 18.3. 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. Buffers may be used for very long transmissions, see Figure 18.4. If so, their overall propagation delay should be stable within the skew tolerance of the link, although the absolute value of the delay is immaterial.

For development support using the standard SGS-THOMSON interfaces the OS-Link should be series terminated as in Figure 18.3.

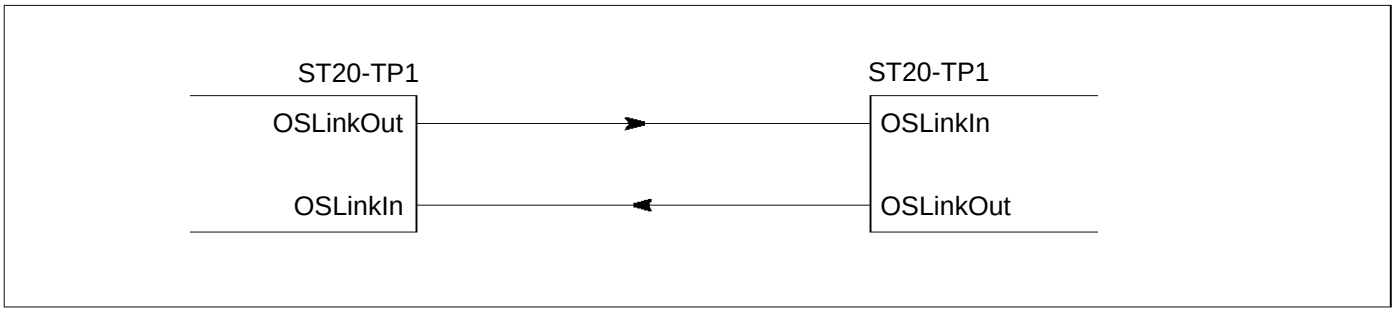


Figure 18.2 OS-Links directly connected

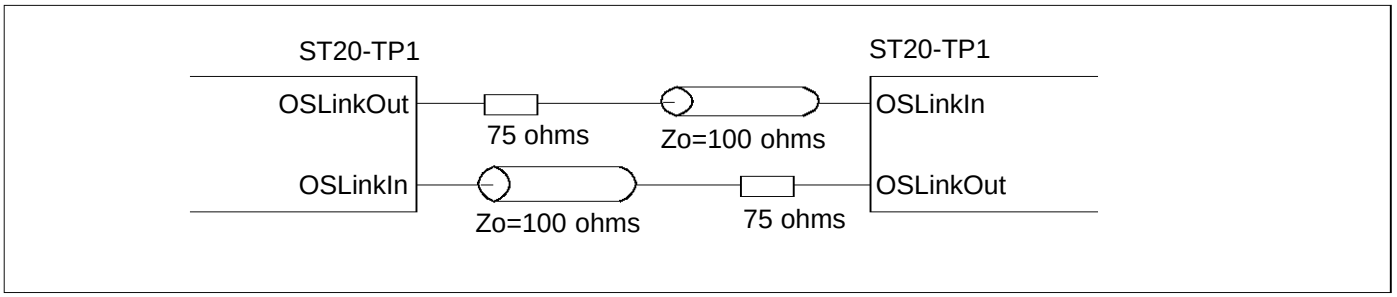


Figure 18.3 OS-Links connected by transmission line

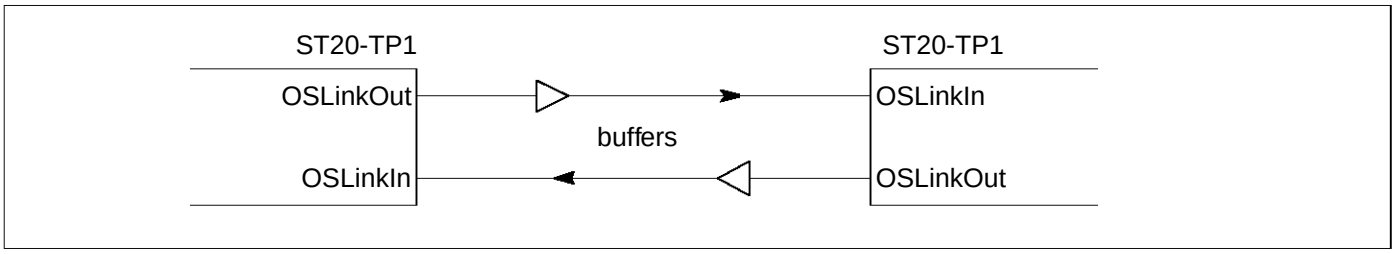


Figure 18.4 OS-Links connected by buffers

19 Link IC interface

The Link-IC interface provides a byte wide data input from the Link-IC. The interface between the CPU and this module is provided using a channel interface as described in A. The base address for the input buffer in the CPU memory space, and the packet size to transfer, are set by the input instruction from the CPU to the Link-IC interface channel. For channel mapping refer to the Memory Map.

All of the signals on the Link-IC external interface are assumed to be synchronous to, and are sampled on, the positive rising edge of the **LByteClk** signal. Data and control signals are clocked into the 24 location input FIFO on each rising edge of **LByteClk** if the **LByteClkValid** signal is high. When the FIFO is full, input data is discarded.

When the software executes an input from the Link-IC module the interface removes data from the input FIFO until a low to high transition of the **LPacketClk** signal is seen. This data byte and all following bytes for which the **LByteClkValid** signal is high are transferred to the memory write FIFO until the programmed packet size has been input. Data is written into memory by the Link-IC interface module as 32-bit words whenever the FIFO level exceeds 4 bytes.

Note, if **LPacketClk** signal is asserted on the data byte present on the end of the input FIFO when the input is executed, the module discards data from the FIFO until another low to high transition of the **LPacketClk** is detected and then starts the transfer.

When the programmed number of bytes have been input the remaining data in the FIFO is written to memory as 32-bit words, where possible, with a part-word write to flush remaining bytes from the FIFO. When all the received data has been written to memory an acknowledge to the channel input is sent to the CPU.

If the **LError** signal is active during the data transfer then the module stops putting data into the memory write FIFO and any data in the FIFO is lost. The base address and count are reset and the link interface module waits for the next active edge of **LPacketClk**. The effect of this is that the packet is lost and the CPU is *not* notified. If the **LError** is active while **LPacketClk** is inactive then the signal is ignored. If **LError** is active on the low to high transition of the **LPacketClk** then the data input is never started and again the module waits for the next packet.

Note to software writers

The Link-IC interface module input FIFO of 24 locations allows a small amount of time for the software to deal with the packet just received and to execute the next input from the Link-IC before data is lost. This allows software to be written which can keep up with the packet rate without data loss.

19.1 External interface

Pin	In/Out	Function
LByteClk	in	Link-IC byte clock
LByteClkValid	in	Link-IC byte clock valid
LData0-7	in	Data input from Link-IC
LError	in	Link-IC packet error
LPacketClk	in	Link-IC packet clock

Table 19.1 Link-IC interface pins

20 MPEG DMA controller

Interfacing to the external application ICs such as the MPEG Audio, MPEG Video or a combined chip is provided in two ways.

- Memory mapped — the device is memory mapped into EMI bank2. The **notCS0-1** strobes are used to provide the chip select strobes needed to access the registers of the IC or ICs.
- DMA output — the MPEG DMA controller can be used to transfer data from memory to a DMA interface on the MPEG controller in response to a request strobe. The MPEG DMA controller transfers the data to a fixed memory address which is decoded by the EMI and causes an access in bank 2 with one of the **notCDSTRB0-1**strobes active.

The interface between the CPU and the MPEG DMA controller is provided using a channel interface as described in A to initiate the DMA transfer. Control registers are provided to allow the characteristics of each to DMA transfer burst in response to a request to be programmed, and the transfer to be suspended. The base address for the output buffer in the memory space and the size of transfer in bytes are set by the output instruction from the CPU to the MPEG DMA controller channel. For channel mapping see the Memory Map.

20.1 External interface

The MPEG DMA module uses the EMI to decode the write address from the DMA controllers to activate the correct **notCDSTRB** signal during an access. The **notCDREQ0-1** are asynchronous signals from the MPEG decoder which request the next burst of data when active.

Pin	In/Out	Function	Notes
notCDREQ0-1	in	Application IC compressed data request	
notCDSTRB0-1	out	Application IC compressed data strobe	1
notCS0-1	out	Application IC chip select	1

Table 20.1 MPEG DMA pins

Notes

1 These signals are common to the EMI and the MPEG DMA interface.

20.2 MPEG DMA transfers

To perform a DMA transfer to an MPEG decoder DMA data port connected to the EMI the MPEG DMA controller must first be initialized via the configuration registers and then an output to the MPEG DMA channel executed by the CPU.

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 DMA port.

After sampling the **notCDREQ** signal active the signal is ignored until the burst size in bytes have been transferred, the last write cycle of the burst has completed, and the hold-off time programmed

in cycles in the **MPEGHoldoff** register 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 should be set to a 0 before a transfer is initiated otherwise the transfer will not start.

Note, the **MPEGBurstSize** and **MPEGHoldoff** registers are not altered by transfer operations and do not have to be set up before each transfer.

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 MPEG DMA controller. Note, if there are less than **BurstSize** bytes left to transfer then only these bytes will be transferred.The destination 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. The EMI will break these word or part word writes into multiple byte writes since the bank width for bank 2 would normally be programmed to be eight bits.

During a transfer DMA operations can be suspended by setting the **MPEGSuspend** register bit to a 1. Note that although no new write transfers will be started after this bit has been set, 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 a 0.

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

The destination address for the data and hence the strobe used as the DMA data strobe are fixed in the two MPEG DMA controllers and are shown in Table 20.2. This table also shows which **notCDREQ** strobe is connected to the MPEG DMA controllers.

MPEG DMA controller	Write address	DMA data strobe	DMA request strobe
0	#00002000	**notCDSTRB0**	**notCDREQ0**
1	#00003000	**notCDSTRB1**	**notCDREQ1**

Table 20.2 MPEG DMA controllers write addresses and strobes

Timings of the **notCDSTRB0-1** lines are programmable via the EMI configuration. See “Support for MPEG application devices” on page 61.

20.3 MPEG configuration registers

The base address for the MPEG DMA configuration registers are given in the Memory Map.

MPEGBurstSize register

The **MPEGBurstSize** register is a write only register and 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 DMA port.

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

Table 20.3 **MPEGBurstSize** register format

MPEGHoldoff register

The **MPEGHoldoff** register is a write only register and must be programmed with the hold-off time from the end of one burst to re-sampling.

MPEGHoldoff	**MPEGDMA base address + #04**	**Write only**
Bit	**Bit field**	**Function**
4:0	**Holdoff4:0**	DMA transfer hold-off time from the end of one burst to re-sampling **notCDREQ0-1**. **Holdoff4:0** — **Hold-off time in system clock cycles** 00000 — 0 cycles 00001 — 1 cycle 00010 — 2 cycles ... — ... 11111 — 31 cycles
7:5		RESERVED. Write 0

Table 20.4 **MPEGHoldoff** register format

MPEGSuspend register

The **MPEGSuspend** register is a write only register and determines whether DMA is enabled (normal operation) or suspended.

MPEGSuspend	**MPEGDMA base address + #08**	**Write only**
Bit	**Bit field**	**Function**
0	**Suspend**	Enables DMA operations 0 — suspends DMA 1 — enables DMA (normal operation)
7:1		RESERVED. Write 0

Table 20.5 **MPEGSuspend** register format

21 DES decryption controller

The Data Encryption Standard (DES) decryption controller (DESC) performs one of two functions:

- decrypting blocks of data
- block moves

In both cases there is an input and an output address and a transfer size that need to be set up. Additionally, a 64-bit decryption key must be provided for the decrypting engine in the case of a decryption transfer.

The interface between the CPU and the DESC is provided using a channel interface as described in A to initiate the DMA transfer. Control registers are provided to allow the mode of operation of the DESC, the transfer destination address for the data, and the DES key to be set up prior to a DMA operation. The base address for the output buffer in the memory space, from which the encrypted data or block move source data is taken, and the size of transfer in bytes are set by the *out* (output) instruction from the CPU to the DMA controller channel. For channel mapping see the Memory Map.

The decrypting engine implements the Data Encryption Standard (DES) Electronic Code Book (ECB) mode decryption algorithm. The algorithm takes a 64-bit input data word, places it in an accumulator and performs an initial bit permutation followed by 16 iterations where the data is combined with a 64-bit decryption key (a 56-bit vector plus 8 bits for per-byte parity). After a final permutation the value remaining on the accumulator is the decrypted data word.

21.1 Decrypting or moving blocks of data

To perform a DMA transfer through the DESC, from one memory buffer to another, the DESC must first be initialized and then an output to the DESC channel executed by the CPU.

The **DESCDest** register must be written with the address of the first byte of the destination buffer before each transfer.

Then the **DESCMode** register bit has to be written to set the mode of operation of the DESC. This register is not altered during a transfer and need not be re-written before each transfer.

If the transfer to be performed is a decryption operation then the 64-bit DES key must be written into the **DESCKeyLSW** and **DESCKeyMSW** registers. Note the parity bits of the DES key are not checked and are discarded. This register is not altered during a transfer and need not be re-written before each transfer.

The final stage of initializing the DESC DMA transfer is to execute an output to the DESC DMA 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 256 bytes.

Note, the decryption algorithm used for DES operates on 64-bit blocks therefore the size of the transfer specified in the *out* instruction to the DESC channel should be a multiple of 8 bytes if decryption is being performed. If decrypting is being performed then the lowest three bits of the byte count must be zeros. Note, for a block move operation the transfer size does not have to be a multiple of 8 bytes.

After the *out* instruction has been executed by the CPU the transfer is started. The DESC DMA controller fetches 64-bit blocks as pairs of words from the source address whenever possible, and either performs DES decryption on blocks of 8 bytes, or just buffers the bytes before performing word writes in pairs to the destination address whenever possible.

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.

21.2 Configuration registers

The base address for the DESC configuration registers are given in the Memory Map.

DESCDest register

The **DESCDest** register must be written with the address of the first byte of the destination buffer before each transfer.

DESCDest	**DESC base address + #00**		**Write only**
Bit	**Bit field**	**Function**	
31:0	**DMADestination**	DMA transfer destination address	

Table 21.1 **DESCDest** register format

DESCMode register

The **DESCMode** register determines the mode of operation of the DES decryption controller.

DESCMode	**DESC base address + #04**		**Write only**
Bit	**Bit field**	**Function**	
0	**Mode**	DESC mode 0 block move 1 decryption	
7:5		RESERVED. Write 0	

Table 21.2 **DESCMode** register format

DESCKeyLSW

The **DESCKeyLSW** register contains the least significant word of the 64-bit DES key for when the DES decryption controller is operating in decryption mode.

DESCKeyLSW	**DESC base address + #08**		**Write only**
Bit	**Bit field**	**Function**	
31:25, 23:17, 15:9, 7:1	**KeyLSW**	DESC 64-bit key least significant word. Note, parity bits in bits 0, 8, 16, 24 are ignored.	
24,16, 8,0		RESERVED. Write 0	

Table 21.3 **DESCKeyLSW** register format

DESCKeyMSW

The **DESCKeyMSW** register contains the most significant word of the 64-bit DES key for when the DES decryption controller is operating in decryption mode.

DESCKeyMSW	**DESC base address + #0C**	**Write only**
Bit	**Bit field**	**Function**
31:25, 23:17, 15:9, 7:1	**KeyMSW**	DESC 64-bit key most significant word. Note, parity bits in bits 0, 8, 16, 24 are ignored.
24,16, 8,0		RESERVED. Write 0

Table 21.4 **DESCKeyMSW** register format

22 Block move DMA

This module copies blocks of data from one byte address to another in memory.

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 transfer in bytes are set by the *out* (output) instruction from the CPU to the DMA controller channel. For channel mapping see the Memory Map.

The interface between the CPU and the block move module is provided using a channel interface as described in A to initiate the DMA transfer.

22.1 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 **BMDmaAddress** register must be written with the address of the first byte of the destination buffer before each transfer. Note, 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 which sets up the source base address and the DMA transfer size. This also de-schedules the software until the transfer is complete.

The maximum transfer size is 65535 bytes.

After the *out* instruction has been executed by the CPU the transfer is started. The block move DMA controller fetches 64-bit blocks as pairs of words from the source address whenever possible, and buffers the bytes before performing word writes in pairs to the destination address.

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.

22.2 Configuration register

BMDmaAddress register

The **BMDmaAddress** register is a write only register and must be written with the first byte of the destination buffer before each transfer. Note, after the transfer this value is left undefined.

BMDmaAddress	**BM base address + #00**	**Write only**
Bit	**Bit field**	**Function**
31:0	**DestAddress**	Address of block move destination.

Table 22.1 **BMDmaAddress** register format

23 High speed data port DMA controller

The high speed data (HSD) port can transfer data from memory to a strobed byte wide data output port using DMA transfers. The output rate of the bytes and the width of the strobe are programmable. Output rates can be up to 7 Mbytes/s.

The interface between the CPU and the HSD port DMA controller is provided using a channel interface, as described in A, to initiate the DMA 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 HSD DMA controller channel.

23.1 External interface

Pin	In/Out	Function
HSData0-7	out	High speed data
HSStrobe	out	High speed data strobe

Table 23.1 High speed data port pins

23.2 High speed data DMA transfers

To perform a DMA transfer to the HSD port the HSD DMA controller must first be initialized via the configuration registers, see Section 23.3, and then an output to the HSD DMA channel executed by the CPU.

The **HSByteInterval** register controls the rate at which bytes are output on the interface after a transfer has started. This needs to be written with a value before a transfer is initiated.

The **HSStrobeDuration** needs to be written to set the strobe width in CPU cycles.

Note, the **HSByteInterval** and **HSByteInterval** registers are not altered by transfer operations and do not have to be set up before each transfer.

The final stage of initializing the DMA transfer is to execute an output to the HSD DMA 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 HSD DMA module will then fetch words where possible from memory and output bytes from the high speed data port.

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.

23.3 High speed data port configuration registers

The HSD port is programmable via configuration registers. These registers can be set by the *devsw* (device store word) instruction.

The base address for the registers are given in the Memory Map.

HSByteInterval register

The value programmed into the **HSByteInterval** register specifies a cycle count in the range 1 to 32 where a programmed value of ‘00000’ specifies a count of 32.

HSByteInterval	**HSD base address + #00**	**Write only**
Bit	**Bit field**	**Function**
4:0	**ByteInterval**	Number of cycles per byte output. Note a value of 00000 specifies a count of 32. **ByteInterval4:0** — **Byte count** 00000 — 32 00001 — 1 00010 — 2 ... — ... 11111 — 31

Table 23.2 **HSByteInterval** register format

HSStrobeDuration register

The strobe can be programmed as 2 or 3 cycle duration.

HSStrobeDuration	**HSD base address + #04**	**Write only**
Bit	**Bit field**	**Function**
0	**StrobeDuration**	Specifies the strobe duration as 2 or 3 cycles. **StrobeDuration0** — **Duration** 0 — 2 cycles 1 — 3 cycles

Table 23.3 **HSStrobeDuration** register format

24 Configuration register addresses

This chapter lists all the ST20-TP1 configuration registers and gives the addresses of the registers. The complete bit format of each of the registers and its functionality is given in the relevant chapter.

The registers can be examined and set by the *devlw* (device load word) and *devsw* (device store word) instructions. Note, they cannot be accessed using memory instructions.

Register	Address	Size	Set	Clear	Read/ Write
HandlerWptr0	#20000000	32			R/W
HandlerWptr1	#20000004	32			R/W
HandlerWptr2	#20000008	32			R/W
HandlerWptr3	#2000000C	32			R/W
HandlerWptr4	#20000010	32			R/W
HandlerWptr5	#20000014	32			R/W
HandlerWptr6	#20000018	32			R/W
HandlerWptr7	#2000001C	32			R/W
TriggerMode0	#20000040	3			R/W
TriggerMode1	#20000044	3			R/W
TriggerMode2	#20000048	3			R/W
TriggerMode3	#2000004C	3			R/W
TriggerMode4	#20000050	3			R/W
TriggerMode5	#20000054	3			R/W
TriggerMode6	#20000058	3			R/W
TriggerMode7	#2000005C	3			R/W
Pending	#20000080	8	Interrupt trigger	Interrupt grant	R/W
Set_Pending	#20000084	8			W
Clear_Pending	#20000088	8			W
Mask	#200000C0	17			R/W
Set_Mask	#200000C4	17			W
Clear_Mask	#200000C8	17			W
Exec	#20000100	8	Interrupt valid	Interrupt done	R/W
Set_Exec	#20000104	8			W
Clear_Exec	#20000108	8			W
ConfigDataField0	#20002000	32			R/W
ConfigDataField1	#20002004	32			R/W
ConfigDataField2	#20002008	32			R/W
ConfigDataField3	#2000200C	32			R/W
ConfigCommand	#20002010	32			W
ConfigStatus	#20002020	32			R

Table 24.1 ST20-TP1 configuration register addresses

Register	Address	Size	Set	Clear	Read/ Write
ASC0BaudRate	#20003000	16			R/W
ASC0TxBuffer	#20003004	16			W
ASC0RxBuffer	#20003008	16			R
ASC0Control	#2000300C	16			R/W
ASC0IntEnable	#20003010	8			R/W
ASC0Status	#20003014	8			R
ASC0GuardTime	#20003018	16			R/W
ASC1BaudRate	#20004000	16			R/W
ASC1TxBuffer	#20004004	16			W
ASC1RxBuffer	#20004008	16			R
ASC1Control	#2000400C	16			R/W
ASC1IntEnable	#20004010	8			R/W
ASC1Status	#20004014	8			R
ASC1GuardTime	#20004018	16			R/W
ASC2BaudRate	#20005000	16			R/W
ASC2TxBuffer	#20005004	16			W
ASC2RxBuffer	#20005008	16			R
ASC2Control	#2000500C	16			R/W
ASC2IntEnable	#20005010	8			R/W
ASC2Status	#20005014	8			R
ASC2GuardTime	#20005018	16			R/W
ScClkVal	#20006000	5			R/W
ScClkCon	#20006004	2			R/W
SSC0BaudRate	#20007000	16			R/W
SSC0TxBuffer	#20007004	16			W
SSC0RxBuffer	#20007008	16			R
SSC0Control	#2000700C	16			R/W
SSC0IntEnable	#20007010	8			R/W
SSC0Status	#20007014	8			R
SSC1BaudRate	#20008000	16			R/W
SSC1TxBuffer	#20008004	16			W
SSC1RxBuffer	#20008008	16			R
SSC1Control	#2000800C	16			R/W
SSC1IntEnable	#20008010	8			R/W
SSC1Status	#20008014	8			R
PWMVal0	#20009000	8			R/W
PWMVal1	#20009004	8			R/W

Table 24.1 ST20-TP1 configuration register addresses

Register	Address	Size	Set	Clear	Read/ Write
CaptureVal0	#20009008	32			R
CaptureVal1	#2000900C	32			R
CaptureControl	#20009010	32			R/W
CaptureStatus	#20009014	8			R
CaptureAck	#20009018	8			W
P0Out	#2000A000	8			R/W
Set_P0Out	#2000A004	8			W
Clear_P0Out	#2000A008	8			W
P0In	#2000A010	8			R
P0C0	#2000A020	8			R/W
Set_P0C0	#2000A024	8			W
Clear_P0C0	#2000A028	8			W
P0C1	#2000A030	8			R/W
Set_P0C1	#2000A034	8			W
Clear_P0C1	#2000A038	8			W
P0C2	#2000A040	8			R/W
Set_P0C2	#2000A044	8			W
Clear_P0C2	#2000A048	8			W
P0Comp	#2000A050	8			R/W
Set_P0Comp	#2000A054	8			W
Clear_P0Comp	#2000A058	8			W
P0Mask	#2000A060	8			R/W
Set_P0Mask	#2000A064	8			W
Clear_P0Mask	#2000A068	8			W
P1Out	#2000B000	8			R/W
Set_P1Out	#2000B004	8			W
Clear_P1Out	#2000B008	8			W
P1In	#2000B010	8			R
P1C0	#2000B020	8			R/W
Set_P1C0	#2000B024	8			W
Clear_P1C0	#2000B028	8			W
P1C1	#2000B030	8			R/W
Set_P1C1	#2000B034	8			W
Clear_P1C1	#2000B038	8			W
P1C2	#2000B040	8			R/W
Set_P1C2	#2000B044	8			W
Clear_P1C2	#2000B048	8			W

Table 24.1 ST20-TP1 configuration register addresses

Register	Address	Size	Set	Clear	Read/ Write
P1Comp	#2000B050	8			R/W
Set_P1Comp	#2000B054	8			W
Clear_P1Comp	#2000B058	8			W
P1Mask	#2000B060	8			R/W
Set_P1Mask	#2000B064	8			W
Clear_P1Mask	#2000B068	8			W
P2Out	#2000C000	8			R/W
Set_P2Out	#2000C004	8			W
Clear_P2Out	#2000C008	8			W
P2In	#2000C010	8			R
P2C0	#2000C020	8			R/W
Set_P2C0	#2000C024	8			W
Clear_P2C0	#2000C028	8			W
P2C1	#2000C030	8			R/W
Set_P2C1	#2000C034	8			W
Clear_P2C1	#2000C038	8			W
P2C2	#2000C040	8			R/W
Set_P2C2	#2000C044	8			W
Clear_P2C2	#2000C048	8			W
P2Comp	#2000C050	8			R/W
Set_P2Comp	#2000C054	8			W
Clear_P2Comp	#2000C058	8			W
P2Mask	#2000C060	8			R/W
Set_P2Mask	#2000C064	8			W
Clear_P2Mask	#2000C068	8			W
P3Out	#2000D000	8			R/W
Set_P3Out	#2000D004	8			W
Clear_P3Out	#2000D008	8			W
P3In	#2000D010	8			R
P3C0	#2000D020	8			R/W
Set_P3C0	#2000D024	8			W
Clear_P3C0	#2000D028	8			W
P3C1	#2000D030	8			R/W
Set_P3C1	#2000D034	8			W
Clear_P3C1	#2000D038	8			W
P3C2	#2000D040	8			R/W
Set_P3C2	#2000D044	8			W

Table 24.1 ST20-TP1 configuration register addresses

Register	Address	Size	Set	Clear	Read/ Write
Clear_P3C2	#2000D048	8			W
P3Comp	#2000D050	8			R/W
Set_P3Comp	#2000D054	8			W
Clear_P3Comp	#2000D058	8			W
P3Mask	#2000D060	8			R/W
Set_P3Mask	#2000D064	8			W
Clear_P3Mask	#2000D068	8			W
MPEG0BurstSize	#2000E000	8			W
MPEG0Holdoff	#2000E004	8			W
MPEG0Suspend	#2000E008	8			W
MPEG1BurstSize	#2000F000	8			W
MPEG1Holdoff	#2000F004	8			W
MPEG1Suspend	#2000F008	8			W
DESCDest	#20010000	32			W
DESCMode	#20010004	8			W
DESCKeyLSW	#20010008	32			W
DESCKeyMSW	#2001000C	32			W
Int0Priority	#20012000	3			R/W
Int1Priority	#20012004	3			R/W
Int2Priority	#20012008	3			R/W
Int3Priority	#2001200C	3			R/W
Int4Priority	#20012010	3			R/W
Int5Priority	#20012014	3			R/W
Int6Priority	#20012018	3			R/W
Int7Priority	#2001201C	3			R/W
Int8Priority	#20012020	3			R/W
Int9Priority	#20012024	3			R/W
Int10Priority	#20012028	3			R/W
Int11Priority	#2001202C	3			R/W
Int12Priority	#20012030	3			R/W
Int13Priority	#20012034	3			R/W
InputInterrupts	#2001203C	14			R
BMDmaAddress	#20014000	32			W
HSByteInterval	#20015000	5			W
HSStrobeDuration	#20015004	1			W

Table 24.1 ST20-TP1 configuration register addresses

25 Pin list

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

States during and after assertion of **notRST** are given for output pins. Codes are as follows: 0 = low; 1 = high; Z = tristate; X = unknown; H = high if not forced from outside (weak pullup).

The state of all pins when the VDD power supply is outside the operating range, or before **notRST** is asserted, is undefined.

Supplies

Pin	In/Out	Function
VDD		Power supply
GND		Ground

Table 25.1 ST20-TP1 supply pins

System

Pin	In/Out	Function
ClockIn	in	System input clock – PLL or TimesOneMode
SpeedSelect0-1	in	PLL speed selector

Table 25.2 ST20-TP1 system services pins

Reset

Pin	In/Out	Function	Reset state	
			During	After
notRST	in	Reset		
CPUReset	in	System reset		
CPUAnalyse	in	Error analysis		
ErrorOut	out	Error indicator	0	0

Table 25.3 ST20-TP1 Reset pins

Interrupts

Pin	In/Out	Function
Interrupt0-3	in	Interrupt

Table 25.4 ST20-TP1 interrupt pins

Link

Pin	In/Out	Function	Reset state	
			During	After
LinkIn	in	Serial data input channel		
LinkOut	out	Serial data output channel	0	0

Table 25.5 ST20-TP1 link pins

Memory

Pin	In/Out	Function	Reset state	
			During	After
MemAddr2-23	out	Address bus	Z	0
MemData0-31	in/out	Data bus. **Data0** is the least significant bit (LSB) and **Data31** is the most significant bit (MSB).	Z	Z
notMemRd	out	Read strobe	0†	1
MemReq	in	Direct memory access request		
MemGrant	out	Direct memory access granted	0†	0
notMemRf	out	Dynamic memory refresh indicator	0†	1
MemWait	in	Memory cycle extender		
notMemCAS0-3	out	CAS strobes – banks 0-3 or bytes 0-3	1	1
notMemRAS0/1/3	out	RAS strobes – banks 0, 1, 3	1	1
notMemPS0/1/3	out	Programmable strobes – banks 0, 1, 3	1	1
notMemBE0-3	out	Byte enable strobes – banks 0-3	1	1
notCS0-1	out	MPEG ICs chip select	1	1
notCDSTRB0-1	out	MPEG ICs compressed data strobe	1	1
BootSource0-1	in	Boot from ROM or from link		
ProcClkOut	out	Processor clock	0	0

†If clocks are running. If clocks are not running the state is unknown.

Table 25.6 ST20-TP1 memory pins

DMA control

Pin	In/Out	Function
notCDREQ0-1	in	MPEG IC compressed data request

Table 25.7 ST20-TP1 DMA control pins

Link IC

Pin	In/Out	Function
LByteClk	in	Link IC byte clock
LByteClkValid	in	Link IC byte clock valid edge
LData0-7	in	Link IC data
LError	in	Link IC packet error
LPacketClk	in	Link IC packet strobe

Table 25.8 ST20-TP1 link IC pins

High speed data port

Pin	In/Out	Function	Reset state	
			During	After
HSData0-7	out	Output data from HSD DMA block	0	0
HSStrobe	out	High speed data strobe	0	0

Table 25.9 ST20-TP1 high speed data port pins

Parallel Input/Output

Pin	In/Out	Function	Reset state	
			During	After
PIO0[0-7]	in/out	Parallel input/output pin or alternate function (see Table 25.11).	H	H
PIO1[0-7]	in/out	Parallel input/output pin or alternate function (see Table 25.11).	H	H
PIO2[0-7]	in/out	Parallel input/output pin or alternate function (see Table 25.11).	H	H
PIO3[0-7]	in/out	Parallel input/output pin or alternate function (see Table 25.11).	H	H

Table 25.10 ST20-TP1 PIO pins

Port bit	Alternate function of PIO pins			
	PIO port 0	**PIO port 1**	**PIO port 2**	**PIO port 3**
0	ASC0 TXD	SSC0 MTSR	ASC2 TXD (ScDataOut)	SSC1 MTSR
1	ASC0 RXD	SSC0 MRST	ASC2 RXD (ScDataIn)	SSC1 MRST
2	-	SSC0 SClk	ScClkGenExtClk	SSC1 SClk
3	-	PWMOut0	ScClk	CaptureIn0
4	-	PWMOut1	(ScRST)	CaptureIn1
5	-	ASC1 TXD	(ScCmdVcc)	CaptureClk0
6	-	ASC1 RXD	ASC2 notOE (ScCmdVpp)	CaptureClk1
7	-	-	(ScDetect)	-

() indicates suggested/possible pin function

Table 25.11 Alternate function of PIO pins

Note: Alternate functions are described in section 29.1.

Test Access Port (TAP)

Pin	In/Out	Function	Reset state	
			During	After
TDI	in	Test data input		
TDO	out	Test data output	Z	Z
TMS	in	Test mode select		
TCK	in	Test clock		
notTRST	in	Test logic reset		

Table 25.12 ST20-TP1 TAP pins

Miscellaneous

Pin	In/Out	Function
NIC		No Connect (this refers to unused pins). Do not wire this pin.

Table 25.13 ST20-TP1 miscellaneous pins

26 Package specifications

The ST20-TP1 is available in a 208 pin plastic quad flat pack (PQFP) package.

26.1 ST20-TP1 package pinout

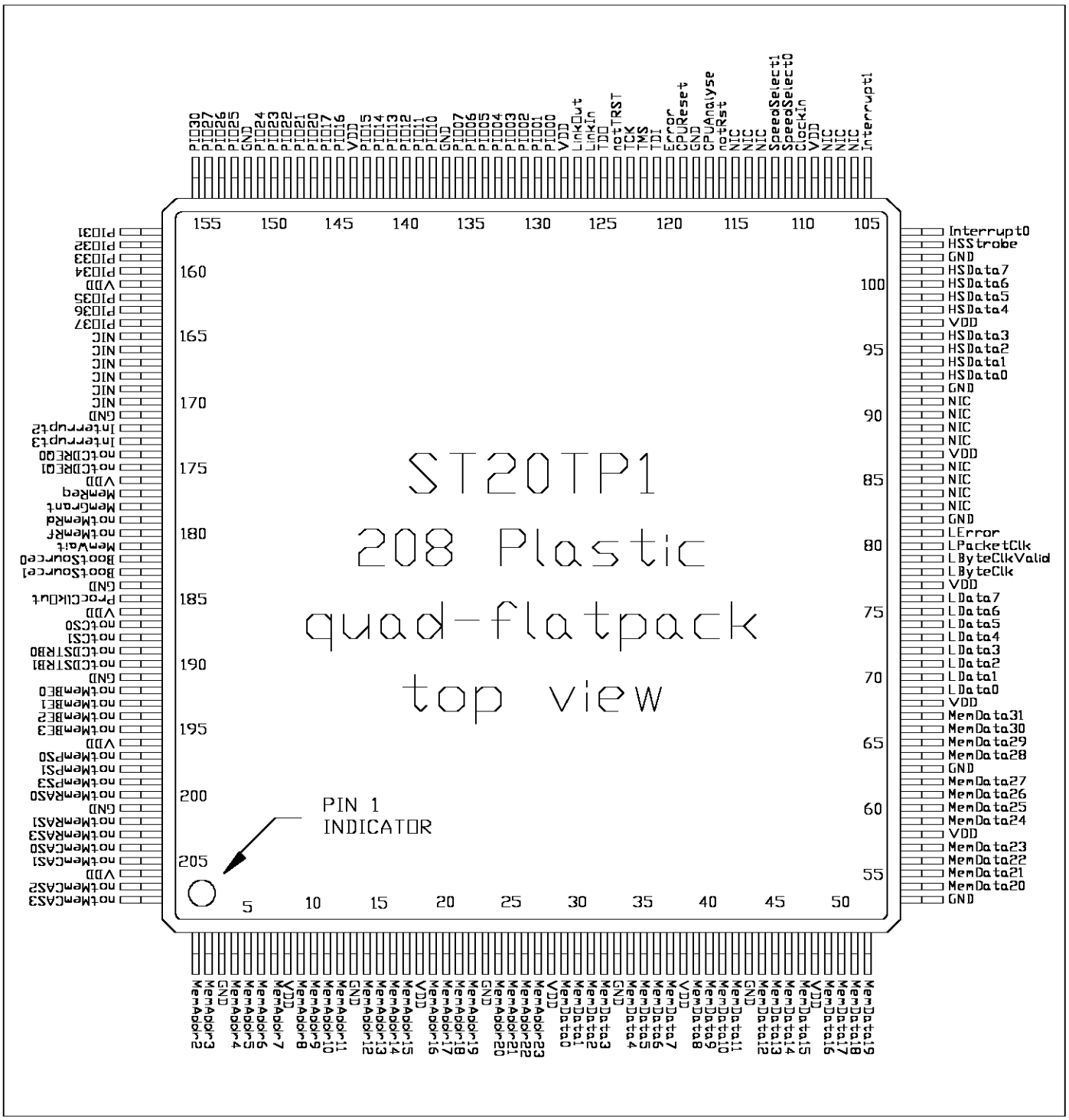


Figure 26.1 ST20-TP1 208 pin PQFP package pinout

26.2 208 pin PQFP package dimensions

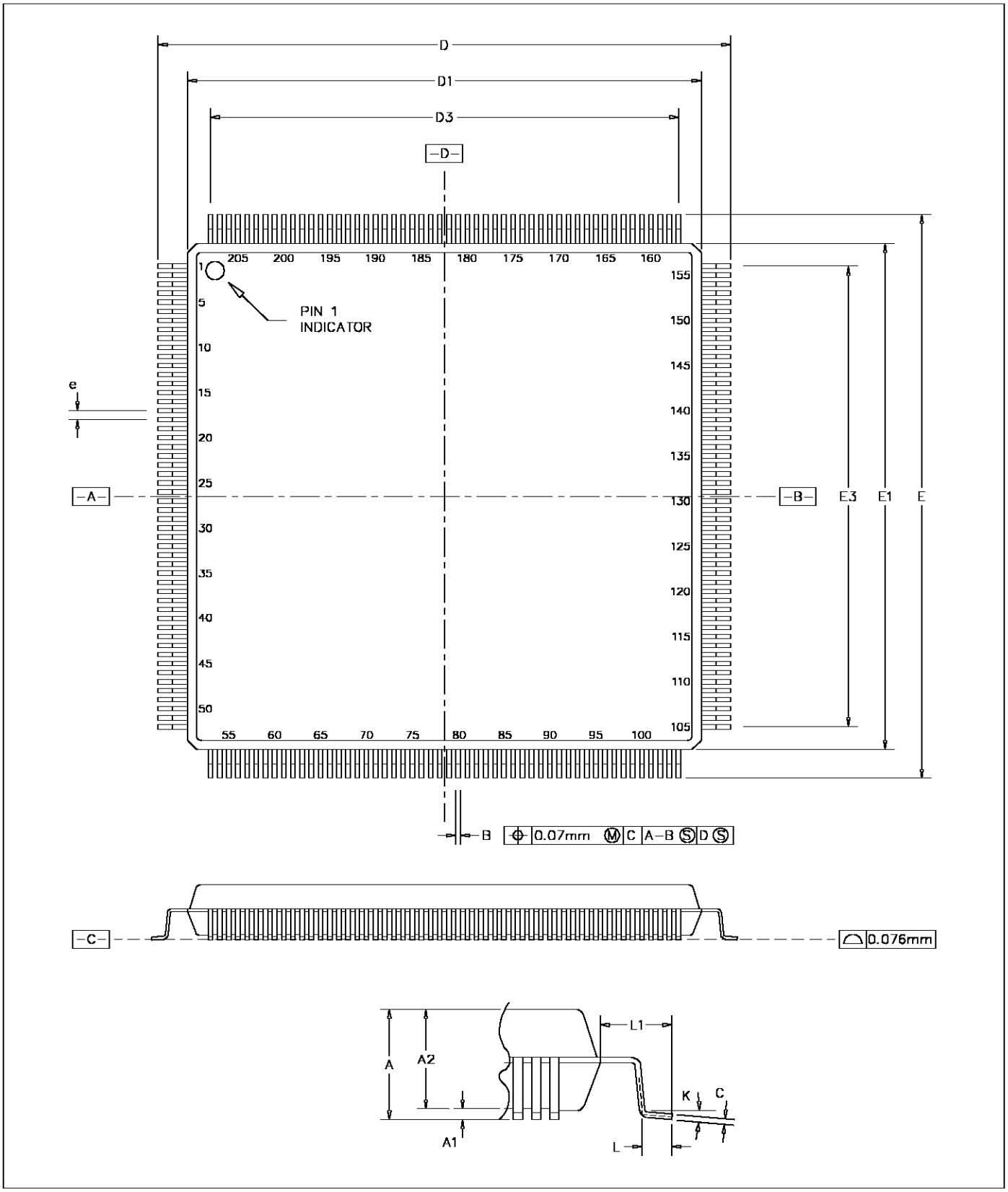


Figure 26.2 208 pin PQFP package dimensions

REF.	CONTROL DIM. mm		
	MIN	NOM	MAX
A			4.10
A1		0.25	
A2	3.40	3.20	3.60
B		0.17	0.27
C		0.09	0.20
D	30.60		
D1	28.00		
D3	25.50		
E	30.60		
E1	28.00		
E3	25.50		
e	0.50		
K	3.5d	0d	7d
L	0.60	0.45	0.75
L1	1.30		

Notes

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

Table 26.1 208 pin PQFP package dimensions

27 Device ID

The identification code for the ST20-TP1 is #*m*5192011, where *m* is a manufacturing revision number reserved by SGS-THOMSON. See Table 27.1.

bit 31																												bit 0
Mask rev		ST20 family						Variant									SGS-THOMSON manufacturers id											a
reserved	0	1	0	1	0	0	0	1	1	0	0	1	0	0	1	0	0	0	0	0	0	0	1	0	0	0	1
		5			1			9			2			0			1			1							

Table 27.1 Identification code

a. Defined as 1 in IEEE 1149.1 standard.

The identification code is returned by the *ldprodid* instruction, see Table 6.4.

28 Ordering information

Device	Package
ST20TP1X40S	208 pin plastic quad flat pack (PQFP)

For further information contact your local SGS-THOMSON sales office.

29 Device configuration

This section gives the assignments of functions to shared pins and the assignment of interrupts to peripherals.

29.1 PIO pins and alternate functions

To allow the flexibility for the ST20-TP1 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 alternate function inputs and outputs of a PIO port bit.

This scheme allows these pins of the device to be configured as general purpose PIO if the associated peripheral input or output is not required in the application.

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

Table 29.1 shows the assignment of the alternate functions to the PIO bits.

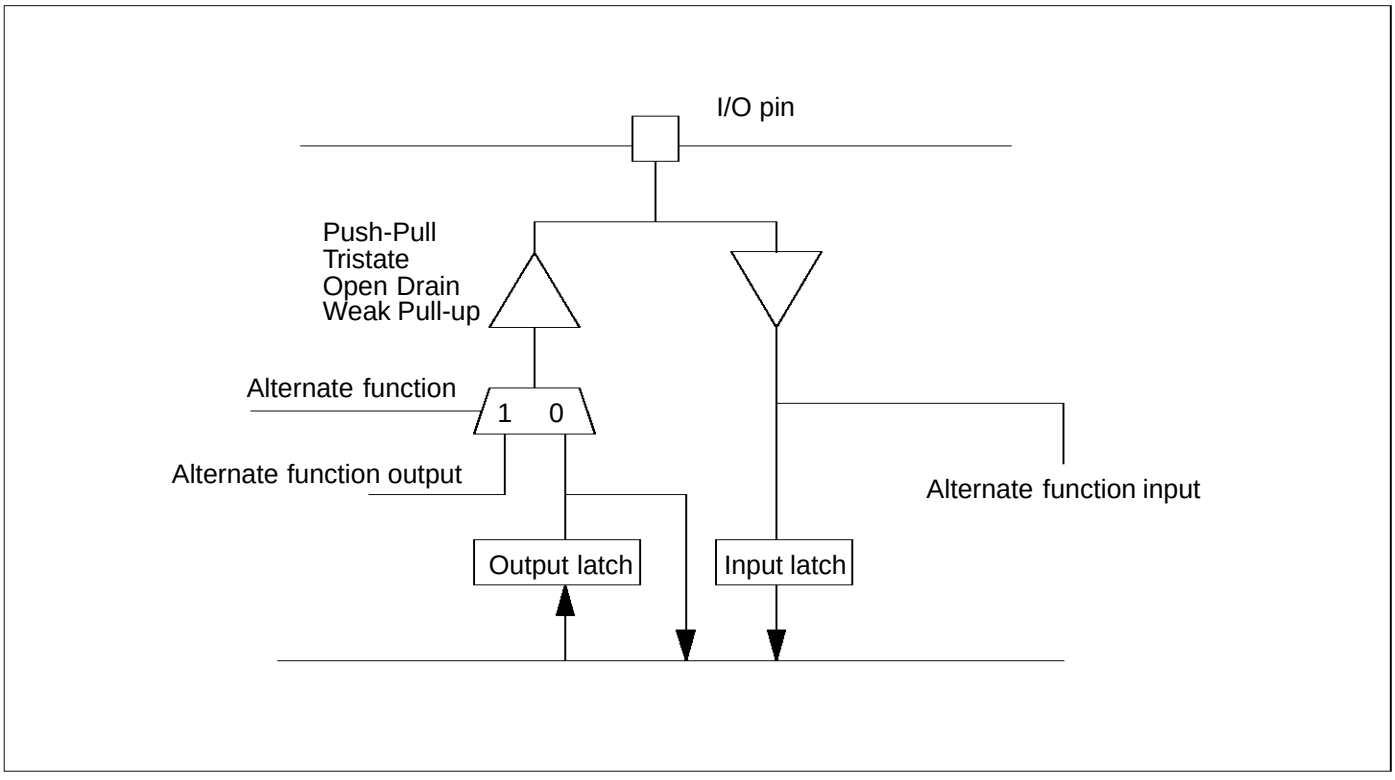


Figure 29.1 I/O port pin

Port bit	Alternate function
Port 0 Bit 0	ASC0 TXD
Port 0 Bit 1	ASC0 RXD
Port 0 Bit 2	-
Port 0 Bit 3	-
Port 0 Bit 4	-
Port 0 Bit 5	-
Port 0 Bit 6	-
Port 0 Bit 7	-
Port 1 Bit 0	SSC0 MTSR
Port 1 Bit 1	SSC0 MRST
Port 1 Bit 2	SSC0 SClk
Port 1 Bit 3	PWMOut0
Port 1 Bit 4	PWMOut1
Port 1 Bit 5	ASC1 TXD
Port 1 Bit 6	ASC1 RXD
Port 1 Bit 7	-
Port 2 Bit 0	ASC2 TXD (ScDataOut)
Port 2 Bit 1	ASC2 RXD (ScDataIn)
Port 2 Bit 2	ScClkGenExtClk
Port 2 Bit 3	ScClk
Port 2 Bit 4	(ScRST)
Port 2 Bit 5	(ScCmdVcc)
Port 2 Bit 6	ASC2 notOE (ScCmdVpp)
Port 2 Bit 7	(ScDetect)
Port 3 Bit 0	SSC1 MTSR
Port 3 Bit 1	SSC1 MRST
Port 3 Bit 2	SSC1 SClk
Port 3 Bit 3	CaptureIn0
Port 3 Bit 4	CaptureIn1
Port 3 Bit 5	CaptureClk0
Port 3 Bit 6	CaptureClk1
Port 3 Bit 7	-

Table 29.1 PIO port alternate function assignments

() indicates suggested/possible pin function

29.2 Interrupt assignments

The interrupts from the peripherals on the ST20-TP1 are assigned as follows:

Interrupt	Peripheral	Signals ORed together to generate interrupt signal
0	Port 0	Compare function
1	Port 1	Compare function
2	Port 2	Compare function
3	Port 3	Compare function
4	SSC0	SSC0TxBufEmpty, SSC0RxBufFull, SSC0ErrorInterrupt
5	SSC1	SSC1TxBufEmpty, SSC1RxBufFull, SSC1ErrorInterrupt
6	ASC2	ASC2TxBufEmpty, ASC2TxEmpty, ASC2RxBufFull, ASC2ErrorInterrupt
7	ASC1	ASC1TxBufEmpty, ASC1TxEmpty, ASC1RxBufFull, ASC1ErrorInterrupt
8	ASC0	ASC0TxBufEmpty, ASC0TxEmpty, ASC0RxBufFull, ASC0ErrorInterrupt
9	PWM and Capture	PWM0Int, PWM1Int, Capture0Int, Capture1Int
10	**Interrupt0** pin	
11	**Interrupt1** pin	
12	**Interrupt2** pin	
13	**Interrupt3** pin	

Table 29.2 Interrupt assignments

These interrupts are inputs to the interrupt level controller, see Chapter 5 on page 33 for details. This allows these interrupts to be assigned to any of eight interrupt priority levels and for multiple interrupts to share a priority level.

30 Electrical specifications

30.1 Absolute maximum ratings

Symbol	Parameter	Min	Max	Units
V_{DDmax}	DC supply voltage		4.5	V
V_{Imax}	Voltage on input and bi-directional pins	GND-0.6	5.75	V
V_{Omax}	Voltage on output pins	GND-0.6	VDD+0.6	V
I_{Omax}	DC output current		25	mA
T_{Smax}	Storage temperature (ambient)	-55	125	°C
T_{Amax}	Temperature under bias (ambient)	-55	125	°C

Table 30.1 Absolute maximum ratings

Note: Stresses greater than those listed under 'Absolute maximum ratings' 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. Exposure to absolute maximum rating conditions for extended periods may affect reliability.

30.2 Operating conditions

Symbol	Parameter	Min	Max	Units	Notes
V_I, V_O	Input or output voltage	0	5.75	V	1
C_L	Load capacitance per pin		60	pF	2
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		1.8	W	3

Table 30.2 Operating conditions

Notes

1 Excursions beyond the supplies are permitted but not recommended.

2 Excluding **LinkOut** load capacitance and EMI pin load capacitance.

3 Measured at 40 MHz with no static loads on the EMI pins and with a 40 pF load on all output pins.

30.3 DC specifications

Symbol	Parameter	Min	Typical	Max	Units	Notes
VDD	Positive supply voltage	3.0	3.3	3.6	V	
V_{IH}	Input logic 1 voltage	2.0		5.75	V	
V_{IL}	Input logic 0 voltage	-0.5		0.8	V	
I_{IN}	Input current (input pin)			±10	µA	1
I_{OZ}	Off state digital output current			±50	µA	2
I_{OZPIO}	Peak off state PIO input/output current			±200	µA	3
I_{OZEMI}	Peak off state EMI input/output current			1	mA	3
I_{WPPIO}	Input weak pull-up current on PIO pins			30	µA	4
V_{OH}	Output logic 1 voltage	2.4			V	5
V_{OL}	Output logic 0 voltage			0.4	V	5
C_{IN}	Input capacitance (input pins)			10	pF	
C_{IO}	Input capacitance (bi-directional pins)			15	pF	
C_{OUT}	Output capacitance			15	pF	

Table 30.3 DC specifications

Notes

1 $0 \le V_I \le 5.5$

2 $0 \le V_I \le V_{DD}$ and $4.5 < V_I < 5.5$

3 $V_{DD} \le V_I \le 4.5V$

4 $0 \le V_I \le V_{DD}$

5 I_{load} = 4 mA for PIO, 2 mA for all other outputs

31 Timing specifications

31.1 EMI timings

The timings are based on the following loading conditions: 40 pF load with the pad drive strengths (refer to EMI chapter for details on the pad drive strength) as follows:

Address pin drive strength at level 1

Strobe pin drive strength at level 1

Data pin drive strength at level 3

The ‘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 is designed to remove process dependent skews from the datasheet description and highlight the dominant influence of address and strobe timings on memory system design.

All timing measurements are taken using an output threshold of 1.5V unless otherwise stated.

The reference clock duty cycle is 40:60 (40 MHz only).

Symbol	Parameter	Min	Max	Units	Note
t_{CHAV}	Reference Clock high to Address valid	-10.0	0.0	ns	
t_{CLSV}	Reference Clock low to Strobe valid	-10.0	2.0	ns	
t_{CHSV}	Reference Clock high to Strobe valid	-10.0	0.0	ns	
t_{RDVCH}	Read Data valid to Reference Clock high	16.0		ns	
t_{CHRDX}	Read Data hold after Reference Clock high	-2.0		ns	
t_{SVRDX}	Read Data hold after Strobe valid	0.0		ns	1
t_{CLWDV}	Reference Clock low to Write Data valid	-10.0	11.0	ns	1
t_{CHWDV}	Reference Clock high to Write Data valid	-10.0	10.0	ns	1
t_{CHRSV}	Reference Clock high to remaining Strobes valid	-7.0	3.0	ns	
t_{CHPH}	Reference Clock high to ProcClkOut high	-7.0	3.0	ns	
t_{WVCH}	MemWait valid to Reference Clock high	16.0		ns	
t_{CHWX}	MemWait hold after Reference Clock high	-2.0		ns	
t_{RVCH}	MemReq valid to Reference Clock high	16.0		ns	
t_{CHRX}	MemReq hold after Reference Clock high	-2.0		ns	
t_{PHWX}	MemWait hold after ProcClkOut high	0.0		ns	1
t_{PHRX}	MemReq hold after ProcClkOut high	0.0		ns	1

Table 31.1 EMI cycle timings

Notes

1 Minimum values are guaranteed by design.

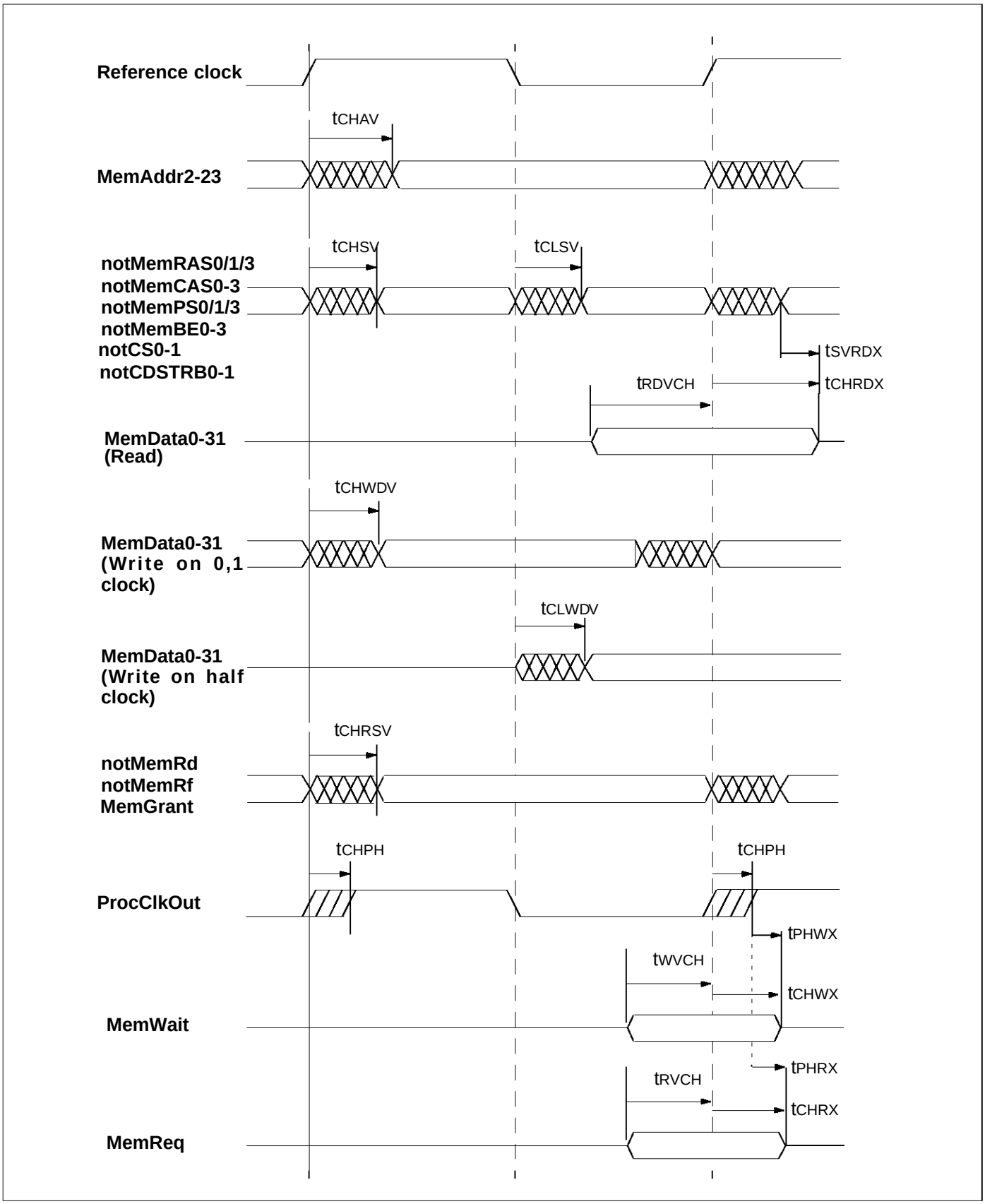


Figure 31.1 EMI timings

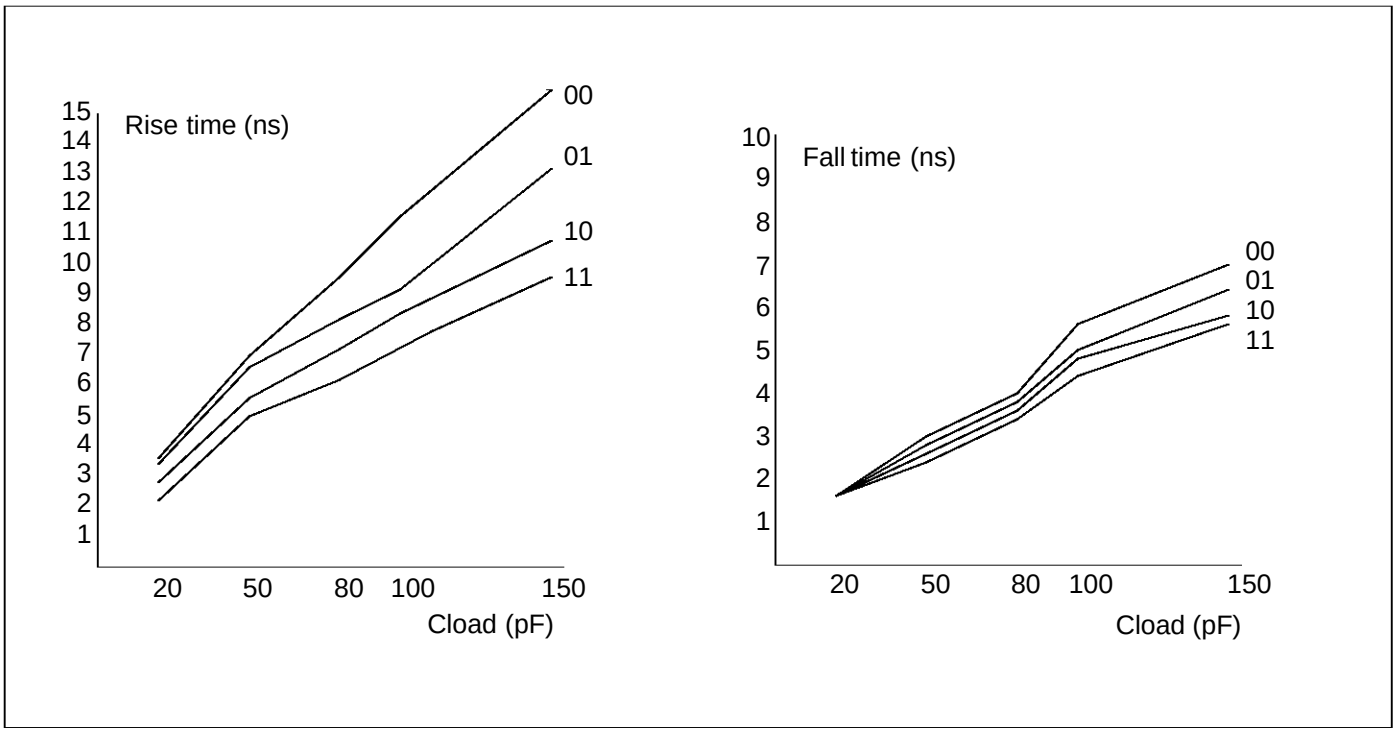


Figure 31.2 Rise and fall times for **data** pins for different pad drive strengths

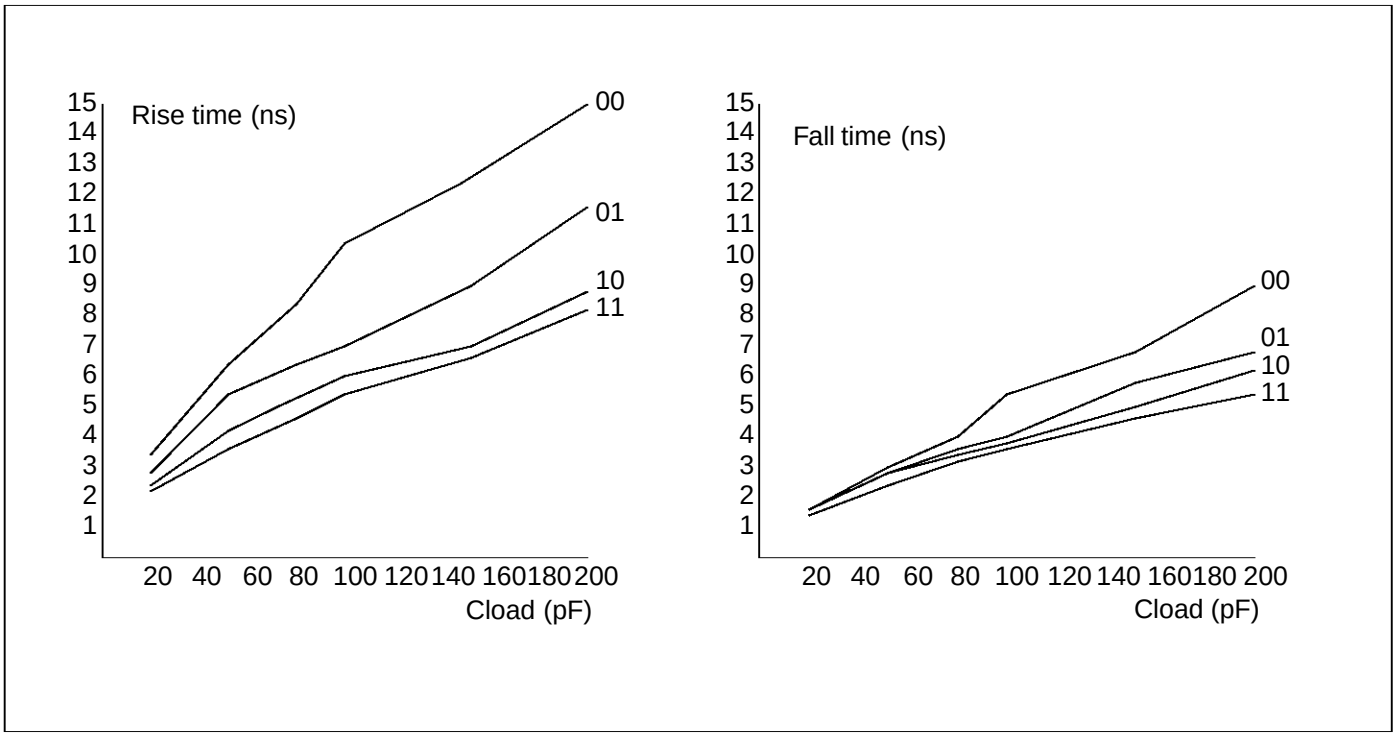


Figure 31.3 Rise and fall times for **address** and **strobe** pins for different pad drive strengths

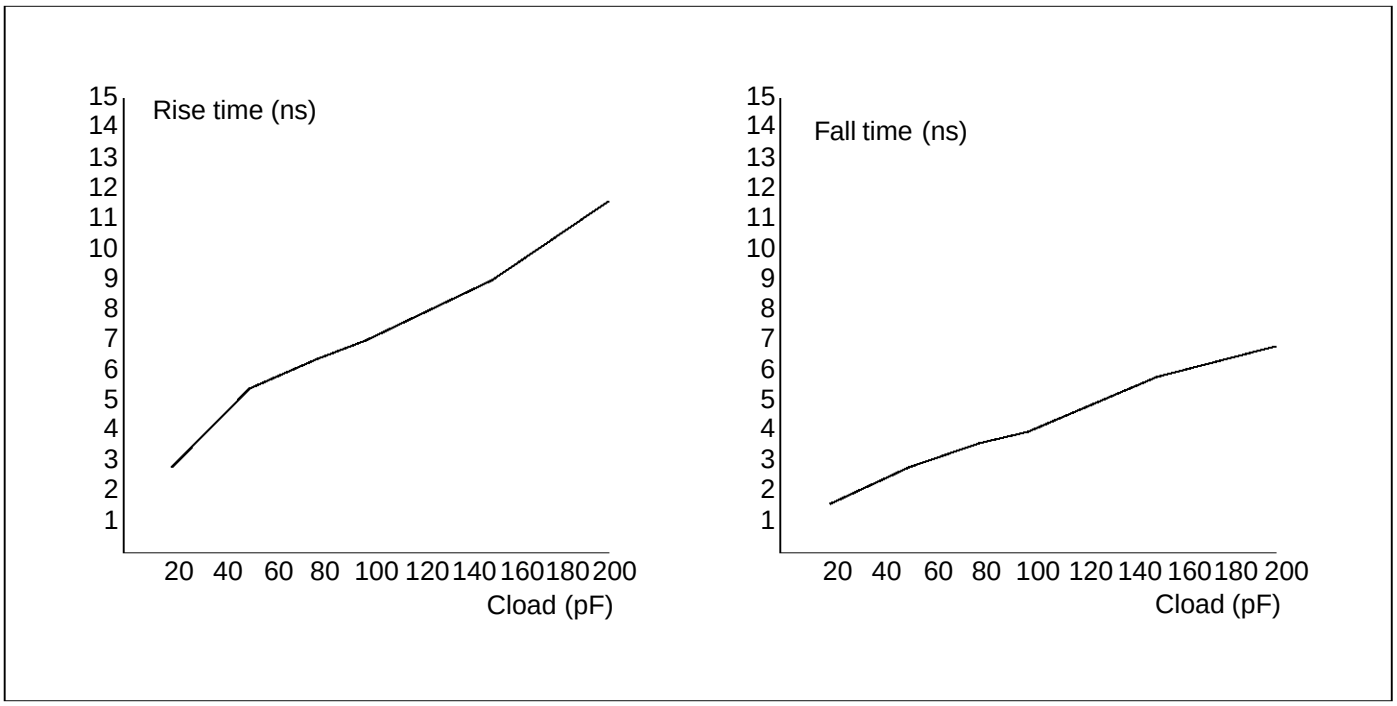


Figure 31.4 Rise and fall times for **ProcClkOut**

All rise and fall times are measured at 10 – 90%, on typical silicon at 3.3 V, 25°C.

31.2 PIO timings

Symbol	No.	Parameter	Min	Max	Units	Notes
t_{IOr}	1	Output rise time	7.0	30.0	ns	1
t_{IOf}	1	Output fall time	7.0	30.0	ns	1

Notes:

1 Load = 50pf.

31.3 Link timings

Symbol	Parameter		Min	Nom	Max	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	
Δt_{JB}	Variation in t_{JQJD}	20 Mbits/s			3	ns	1
		10 Mbits/s			10	ns	1
		5 Mbits/s			30	ns	1
C_{LIZ}	**LinkIn** capacitance	@ f=1MHz			10	pF	
C_{LL}	**LinkOut** load capacitance				50	pF	

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.

Table 31.2 Link timings

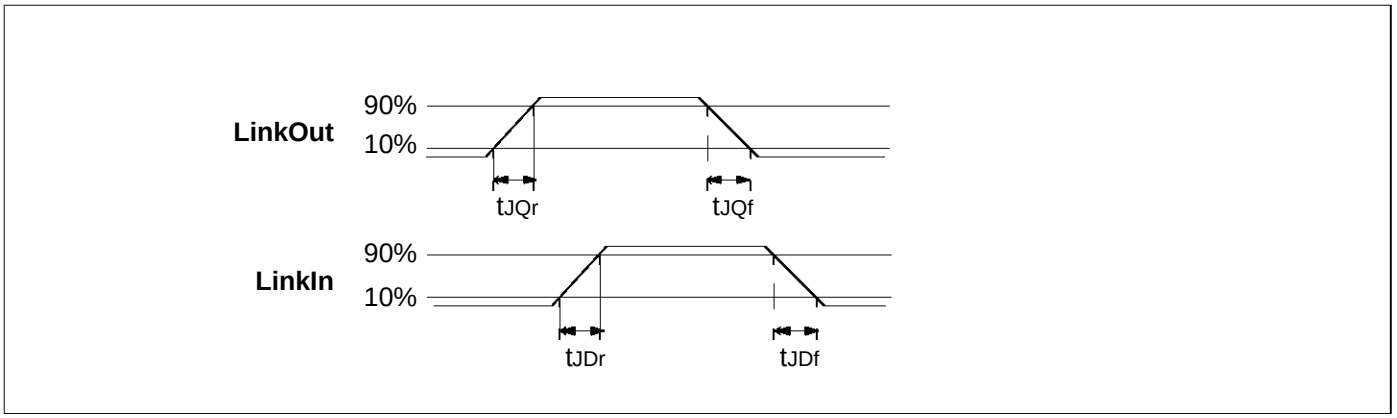


Figure 31.5 Link timings

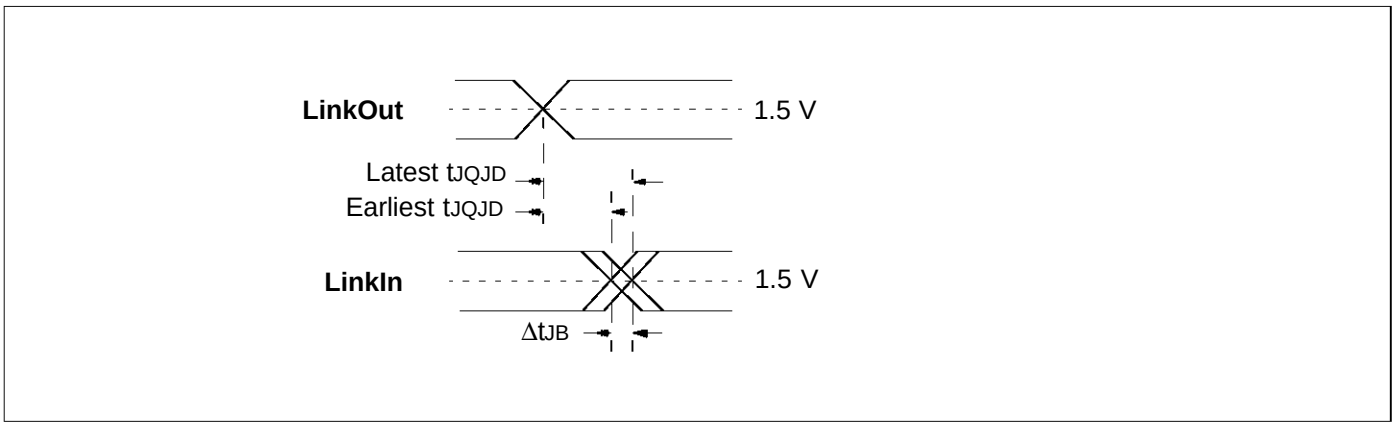


Figure 31.6 Buffered Link timings

31.4 Reset and Analyse timings

Symbol	Parameter	Min	Nom	Max	Units
tRHRL	**notRST** pulse width low	8			ClockIn
tRHRL	**CPUReset** pulse width high	1			ClockIn
tAHRH	**CPUAnalyse** setup before **CPUReset**	3			ms
tRLAL	**CPUAnalyse** hold after **CPUReset** end	1			ClockIn

Table 31.3 Reset and Analyse timings

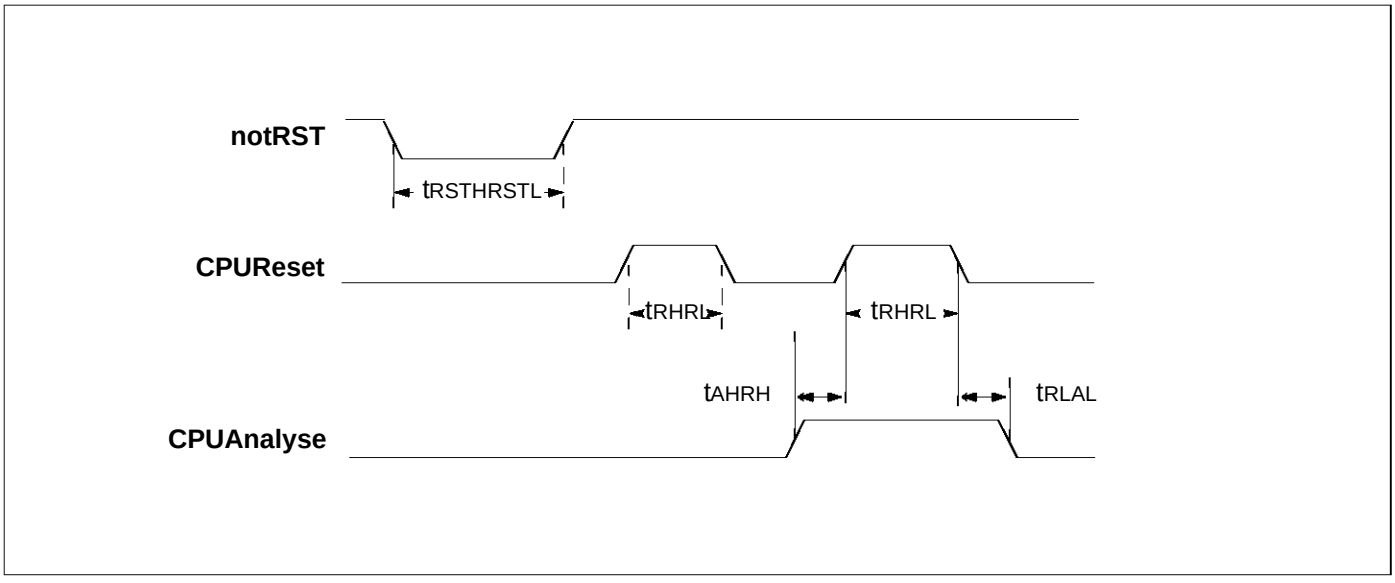


Figure 31.7 Reset and Analyse timings

31.5 Clock timings

31.5.1 ClockIn timings

Symbol	Parameter	Min	Nom	Max	Units	Notes
t_{DCLDCH}	**ClockIn** pulse width low	6			ns	
t_{DCHDCL}	**ClockIn** pulse width high	10			ns	
t_{DCLDCL}	**ClockIn** period		37		ns	1, 2
t_{DCr}	**ClockIn** rise time			10	ns	3
t_{DCf}	**ClockIn** fall time			10	ns	3

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} (see Electrical Specifications chapter).

Table 31.4 **ClockIn** timings

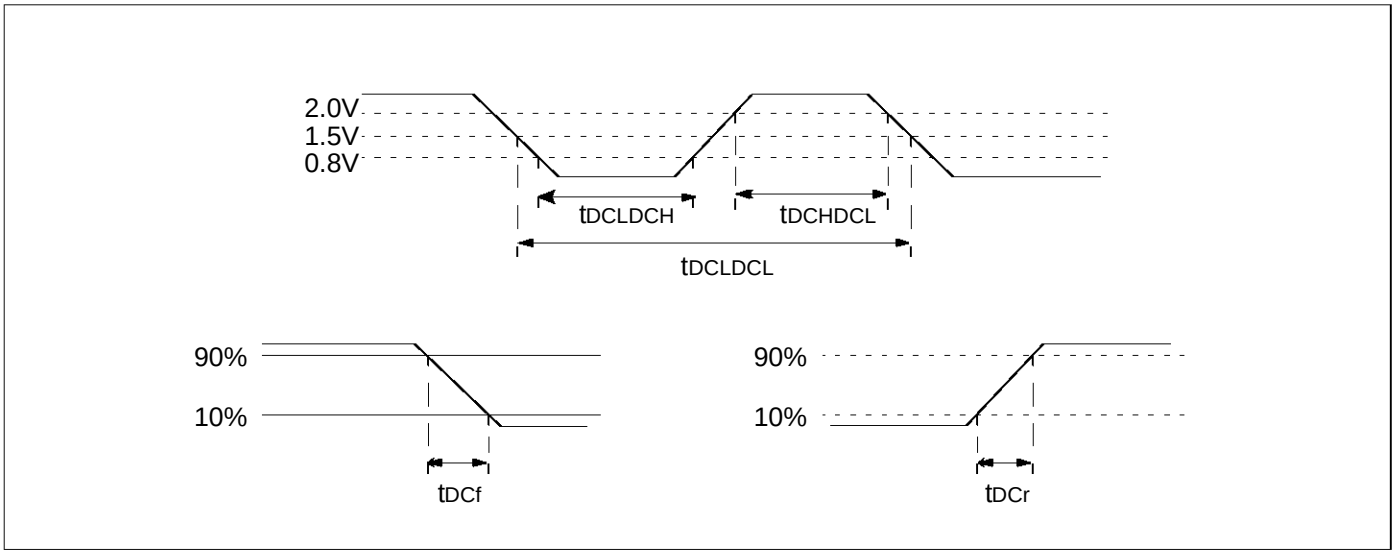


Figure 31.8 **ClockIn** timings

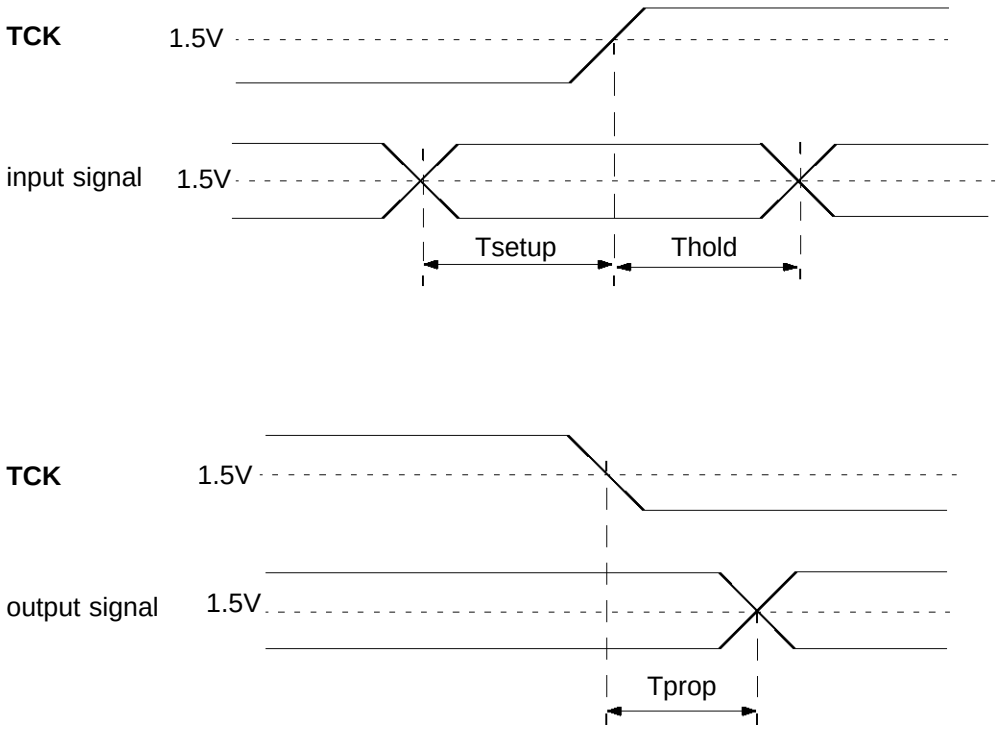
31.6 TAP timings

The TAP will function at 5 MHz **TCK** with the following timings. All other electrical characteristics of the TAP pins are as defined in the Electrical Specifications chapter.

Symbol	Parameter	Min	Nom	Max	Units
Tsetup	Set-up time	10			ns
Thold	Hold time	10			ns
Tprop	Propagation delay			50	ns

Table 31.5 TAP timings

Figure 31.9 TAP timings



31.7 Link IC timings

Symbol	Parameter	Min	Nom	Max	Units	Notes
t_{LCLLCL}	**LByteClk** period	100			ns	
t_{LCHLCL}	**LByteClk** pulse width high	10			ns	
t_{LCLLCH}	**LByteClk** pulse width low	10			ns	
t_{LDVLCH}	Link IC signal valid to **LByteClk** high	10			ns	
t_{LCHLDX}	Link IC signal hold after **LByteClk** high	3			ns	

Table 31.6 Link IC timings

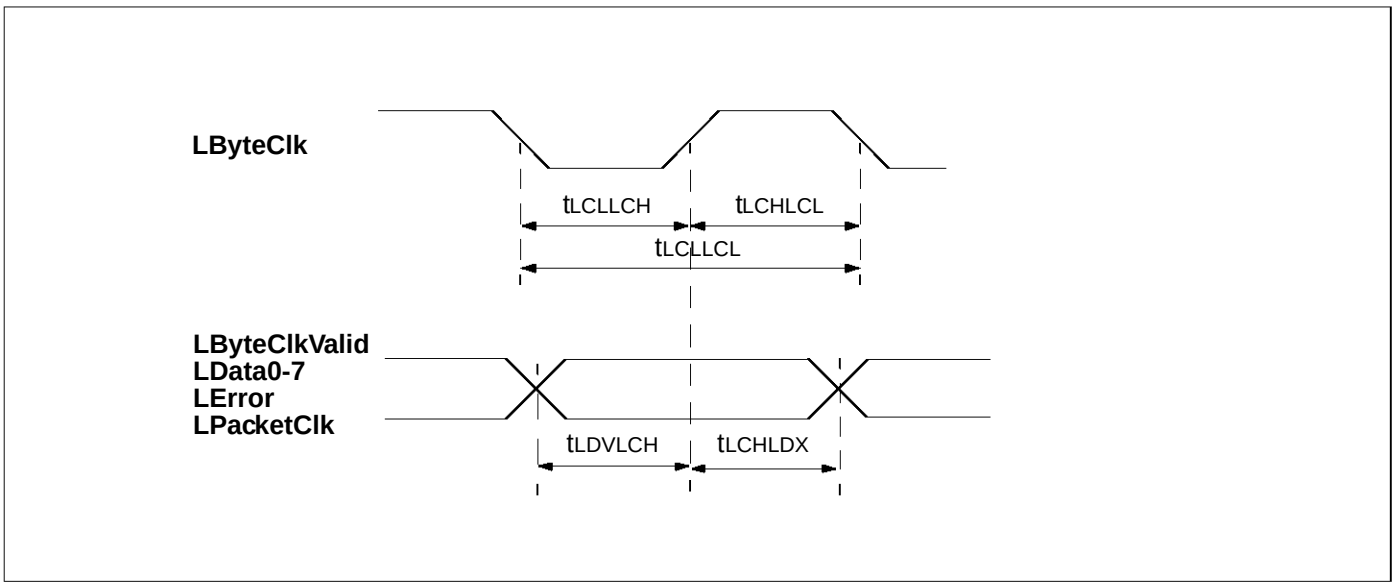


Figure 31.10 Link IC timings

31.8 High speed data port DMA timings

Symbol	Parameter	Min	Nom	Max	Units
t_{HSHHSL}	HSStrobe duration				ns
	for programmed strobe duration of 2 cycles	40	50	60	
	for programmed strobe duration of 3 cycles	65	75	85	
t_{HSHHSH}	HSStrobe byte interval				ns
	for programmed byte interval of 6	148	150	152	
t_{HSHHDV}	HSData valid after HSStrobe high	-10	0	10	

Table 31.7 High speed data port DMA timings

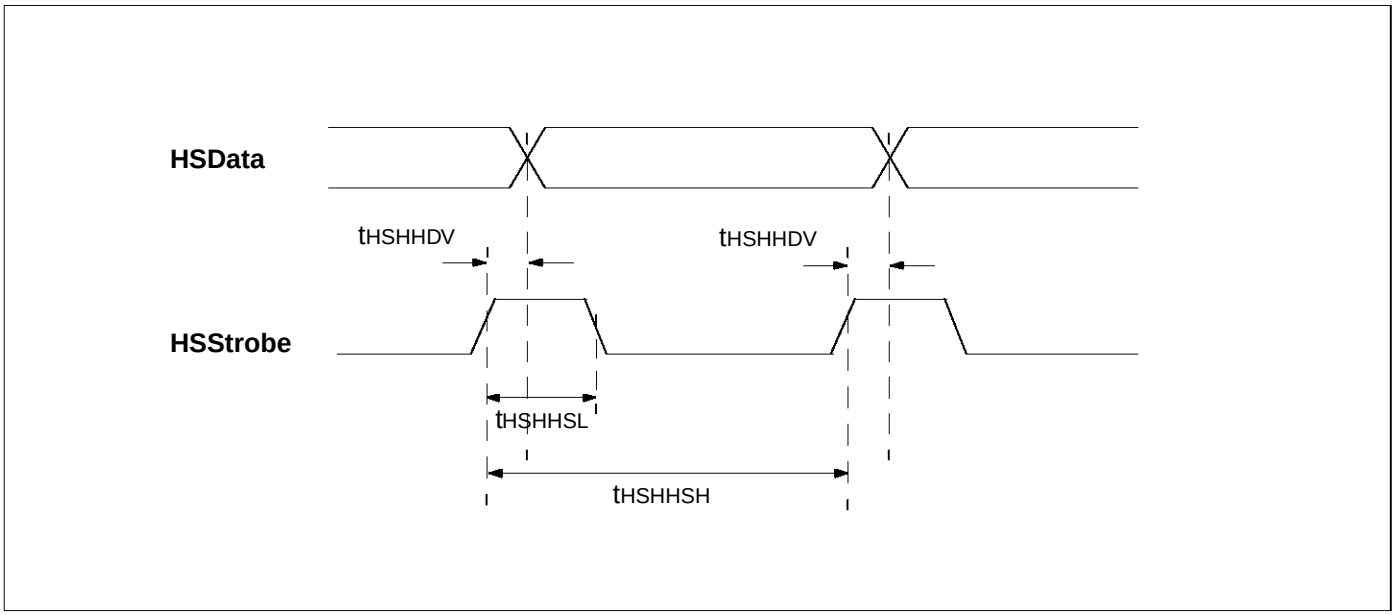


Figure 31.11 Data port timing diagram

A Channel model

The ST20-TP1 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* (input) instruction from the Link-IC interface 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 #8000002C. 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 Link-IC interface 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 #8000002C and adds it to the back of the active list.