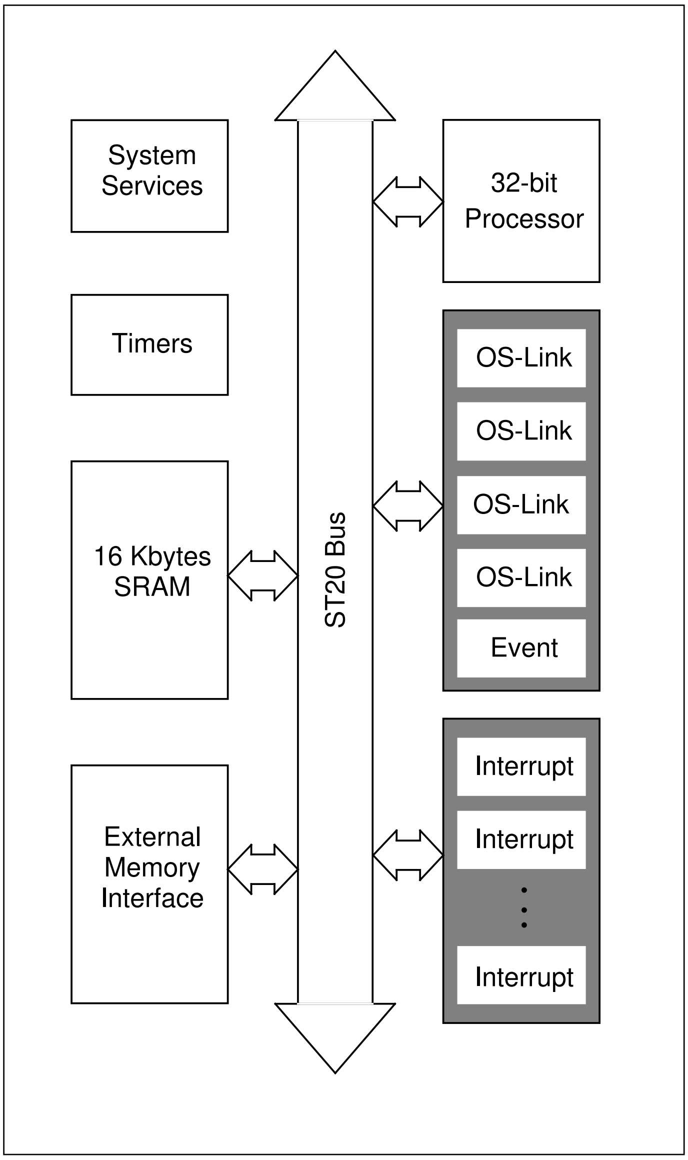
ST20450

32 BIT MICROPROCESSOR

ENGINEERING DATA

FEATURES

- Enhanced 32-bit CPU
  - 0 to 40 MHz processor clock
  - 32 MIPS at 40 MHz
  - fast integer/bit operations
- 16 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
- Serial communications
  - 4 OS-Links
  - 5/10/20 Mbits/s Link0, 10/20 Mbits/s Link1-3
  - Event channel
- Vectored interrupt subsystem
  - Fully prioritized interrupts
  - 8 levels of preemption
  - 500 ns response time
- Power management
  - low power operation
  - power down mode
- Professional toolset support
  - ANSI C compiler and libraries
  - INQUEST advanced debugging tools
- Technology
  - 0 to 40 MHz processor clock
  - 0.5 micron process technology
  - 3 V operation (3 V outputs/bi-directionals, 5 V inputs)
- 208 pin PQFP package
- Test Access Port



APPLICATIONS

- Global positioning by satellite (GPS) receivers
- ISDN terminals
- ATM networks
- Set top terminals
- Industrial control
- Imaging systems

Trap enables word

This register contains the word which encodes the trap enable and global interrupt masks. This should be ANDed with the existing masks to allow the trap handler to disable various events while it runs. However, currently the trap enables word overwrites the existing enables.

With this implementation a trap must disable all other trap groups to ensure correct execution, this increases the difficulty of writing nested trap handlers.

Analysing high priority processes

If the machine is running a high priority process before being analysed and the device is wired to boot from link, not ROM, then the machine will resume execution of that high priority process as soon as it is released from analyse. The **Wptr** will also be set back to **MemStart**.

This will cause the machine to crash, there is no workaround.

Disable external micro-interrupt requests

There are four possible combinations of enable/disable for high and low priority micro-interrupts, however only three are supported. It is not recommended to disable high priority micro-interrupts with low priority interrupts enabled.

With this combination it is possible for high priority external events that are apparently disabled to be accepted as low priority transactions.

Timer scheduler traps always run high priority handler

When a scheduler trap is set, the priority of the scheduler event is stored in the scheduler trap priority field of the **Status** register. For *Timer* scheduler traps the priority is always recorded as high.

This results in the wrong trap handler being executed, which is fatal if there is no high priority scheduler trap installed.

Incorrect order of boundary-scan chain

Due to an error in the boundary-scan logic, **EventWaiting** effectively does not have a boundary-scan cell.

The consequence being that EXTEST will no longer function correctly for the **EventWaiting** pin. Revised boundary scan description language (BSDL) is available.

Illegal configuration for strobes

If any strobe is configured with edge times of: e1time = 0, e2time = (castime*2) - 1 phases, that strobe will behave erroneously when back to back accesses occur.

Erroneous address during RAS time for 16/8 bit accesses

An erroneous address may appear during the first cycle of RAS time whenever a multi-access transaction includes an excursion into a precharge state, for example, if a series of 8-bit accesses are interrupted by a refresh.

The solution is to program the **ShiftAmount** for all banks with the same value. Memory banks consisting of SRAM only will not be affected by the **ShiftAmount** value as it is not used unless RAS time is non zero anyway.

Divide and remainder instruction errors

The *div* (divide) and *rem* (remainder) instructions do not work correctly in some situations if the dividend is negative.

The compiler works around this by replacing each *div* and *rem* instruction with a code sequence which first checks the sign of the dividend and, if it is negative, flips the sign, performs the operation and then flips the sign of the result.

Where these instructions are used explicitly in other toolset software, e.g. libraries, the workarounds have been hard coded. Unlike compiler generated workarounds, it is not possible to disable such hard coded workarounds.

Contents

1 Introduction

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 micros at the lowest cost and fastest time to market. The ST20 macrocell library includes the ST20 family of 32-bit micro cores, embedded memory, standard peripherals, I/O, controllers and customer specified ASICs.

Using ST20 macrocell technology SGS-THOMSON's world-wide technology partners in markets such as set top boxes, digital cellular handsets, hard disk drives and laser printers, are able to specify optimized 32-bit microcontrollers for their applications. Using ST20 technology SGS-THOMSON is able to develop these low cost application specific micros, from paper specification to silicon in less than six months.

The ST20 family of 32-bit micro cores has been designed for applications ranging from deeply embedded low cost portable systems to high performance applications requiring DSP type performance. At the heart of each ST20 core is a highly efficient 32-bit RISC processor, running at frequencies up to 40 MHz on 0.5 micron technology, and achieving up to 32 MIPS performance. The ST20 RISC CPU has a very efficient instruction set and achieves ultra high code density minimizing system ROM requirements and reducing system cost. The first member of the core family, the ST20C4, is a high performance core incorporating the ST20 RISC CPU, the ST20 in-core microkernel and the ST20 high performance arithmetic accelerator. The arithmetic accelerator contains a dedicated hardware multiplier providing two cycle 32-bit multiply and a hardware barrel shifter providing single cycle bit shift. The ST20 in-core microkernel is the highest performance micro-kernel of any 32-bit core, directly supporting multi tasking, I/O, DMA, interrupts, trap handling and timers. It combines the flexibility of a software RTOS with the performance of a microcoded scheduler, achieving context switch and interrupt response times of less than 500ns. It also provides a platform for the efficient porting of industry standard run-time kernels and operating systems (RTOS), making the ST20 ideal for interrupt driven real-time applications that require high data throughput, combined with high performance data processing.

The first product developed using the ST20 macrocell library is the ST20450, incorporating the ST20C4 core, 16 Kbytes of on-chip memory for fast access to local code, a vectored interrupt controller, serial I/O links, and a multi-bank external memory controller supporting SRAM, DRAM, ROM and memory mapped peripherals. The ST20450 has been designed for a class of applications that require high CPU performance and real time execution coupled with low power operation, including ISDN terminals, ATM networks and industrial control systems. Furthermore, the ST20450 acts as a reference platform for all future ST20 application specific designs.

To support the fast turnaround development of ST20 application specific micros an integrated software and hardware development environment has been developed to enable efficient, high performance code to be written and debugged, and system hardware to be simulated, developed and tested. This includes a complete professional ANSI-C software toolset, and the INQUEST window based debugging tools supporting PC and UNIX hosts.

2 Architecture

Figure 2.1 shows the subsystem modules that comprise the ST20450. These modules are outlined below and more detailed information is given in the following chapters of this datasheet.

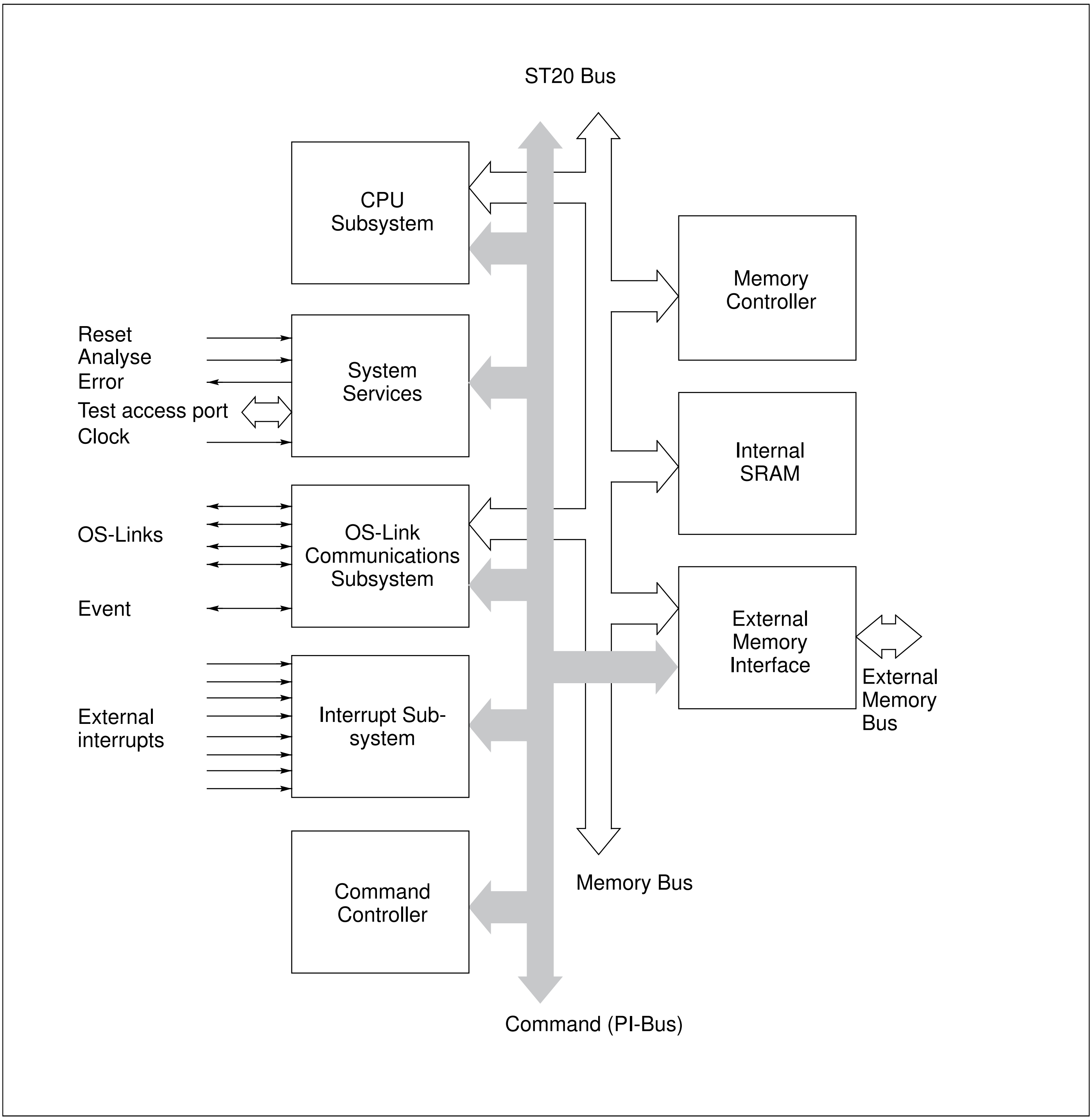


Figure 2.1 ST20450 architectural block diagram

The modules are connected via the ST20 Bus. The ST20 Bus is based on two buses, the memory bus and the command bus. The memory bus is used to access all memory in an ST20450 system, both internal and external. It supports single cycle pipelined accesses with two cycle latency. The command bus is used for interfacing to standard peripherals and sending control information between ST20450 subsystems. The command bus supports point to point message passing using channel communications, and supports the PI-Bus protocol whereby a subsystem can directly access a memory mapped peripheral.

CPU

The Central Processing Unit (CPU) on the ST20450 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 ST20450 on-chip memory system provides 200 Mbytes/s internal data bandwidth, supporting pipelined 2-cycle internal memory access at 25 ns cycle times. The ST20450 memory system consists of SRAM and an external memory interface (EMI). The first ST20450 product has 16 Kbytes of on-chip SRAM.

ST20450 derivative products will have a minimum of 4 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.

The ST20450 External Memory Interface (EMI) controls the movement of data between the ST20450 and off-chip memory. It is designed to support memory subsystems with minimal (often zero) external support logic and is programmable to support a wide range of memory types.

The ST20450 EMI can access a 4 Gbyte physical address space, and provides sustained transfer rates of up to 100 Mbytes/s for SRAM, up to 89 Mbytes/s using page-mode DRAM.

Communications subsystem

The ST20450 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). Each OS-Link provides a pair of channels, one input and one output channel.

There are four OS-Links (**Link0-3**) on the ST20450 which act as individual DMA engines independent of the CPU. The OS-Links have programmable unidirectional data rates of 10 Mbits/s or 20 Mbits/s, giving a bi-directional bandwidth of 3 Mbytes/s. **Link0** can also be run at 5 Mbits/s. The links are used for:

- interfacing to external peripherals
- bootstrapping
- debugging

Interrupt subsystem

The ST20450 interrupt subsystem supports eight prioritized interrupts. This allows nested pre-emptive interrupts for real-time system design.

System services module

The ST20450 system services module includes:

- reset, initialization and error port.
- phase locked loop (PLL) - accepts 5 MHz input and generates all the internal high frequency clocks needed for the CPU and the OS-Links.
- test access port.

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

The scheduler provides a single level of pre-emption. In addition, multi-level pre-emption is provided by the interrupt subsystem, see Chapter 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.

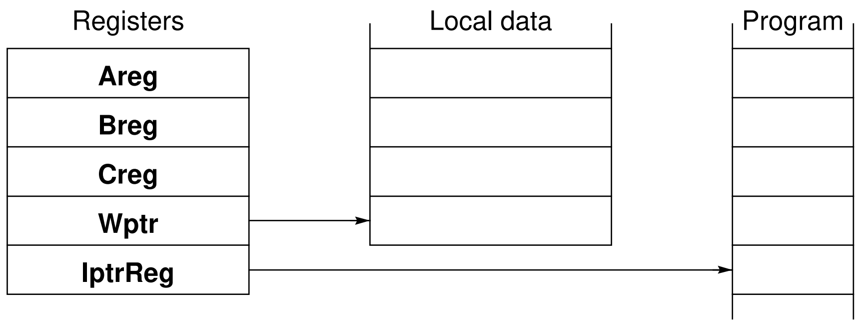


Figure 3.1 Registers used in sequential integer processes

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 portion of the processor's time to each active low priority process in turn (see Section 3.3 on page 14). 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 3.2, 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.

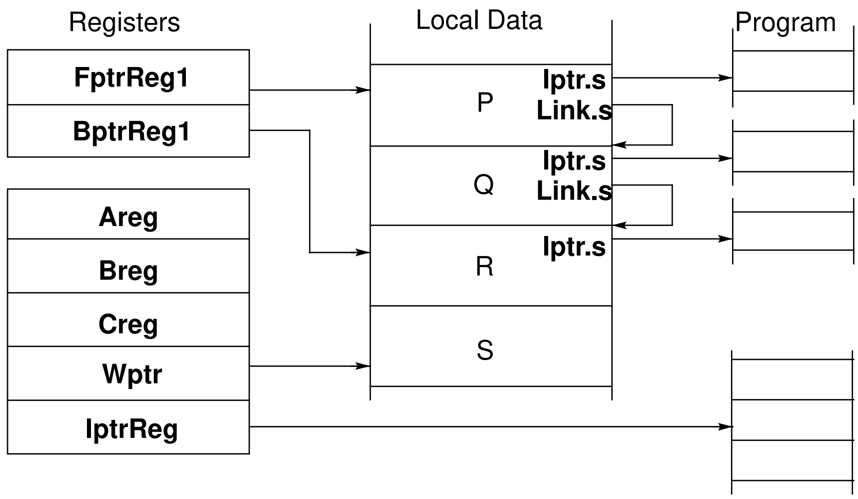


Figure 3.2 Linked process list

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 construct, *startp* is used to create the necessary additional concurrent processes. A *startp* instruction creates a new process by adding a new workspace to the end of the scheduling list, enabling the new concurrent process to be executed together with the ones already being executed. When a process is made active it is always added to the end of the list, and thus cannot pre-empt processes already on the same list.

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

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 re-loaded from the shadow registers and the interrupted low priority process continues executing. Instructions are provided on the processor core to allow a high priority process to store the shadow registers to memory and to load them from memory. Instructions are also provided to allow a process to exchange an alternative process queue for either priority process queue (see Table 5.21 on page 36). 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 every microsecond, cycling completely in approximately 4295 seconds. The other is accessible only to low priority processes and is incremented every 64 microseconds, giving 15625 ticks in one second. It has a full period of approximately 76 hours. All times are approximate due to the clock rate.

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

Note, these timers stop counting when power-down mode (see Chapter 11) is invoked.

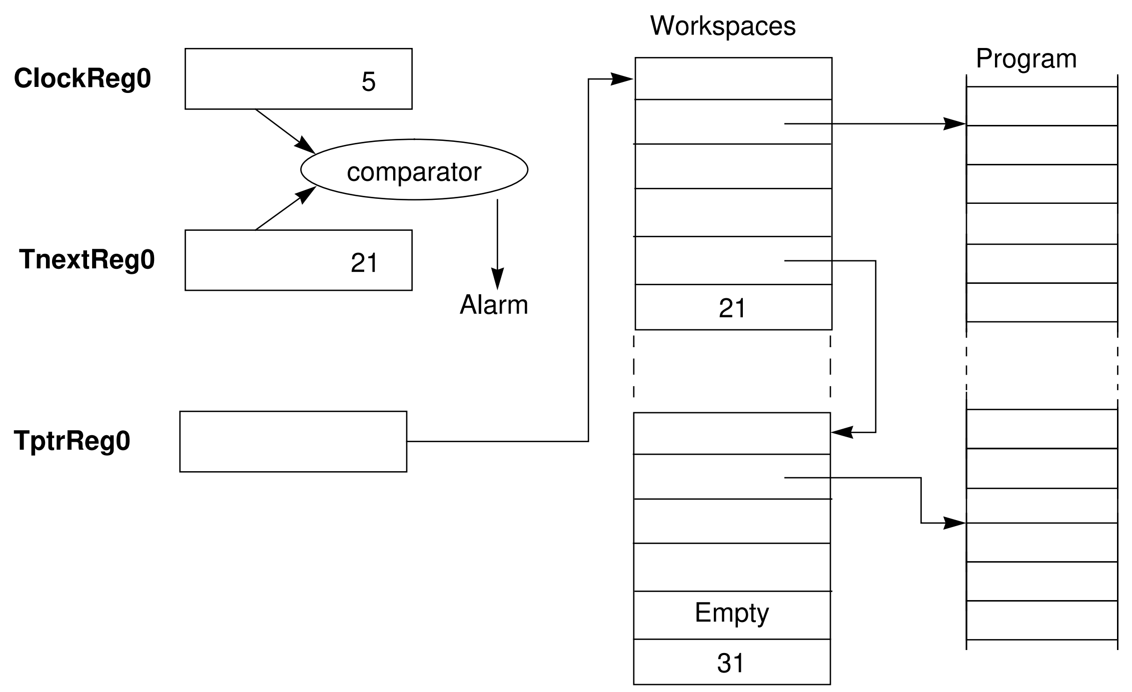


Figure 3.3 Timer registers

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 3.4.

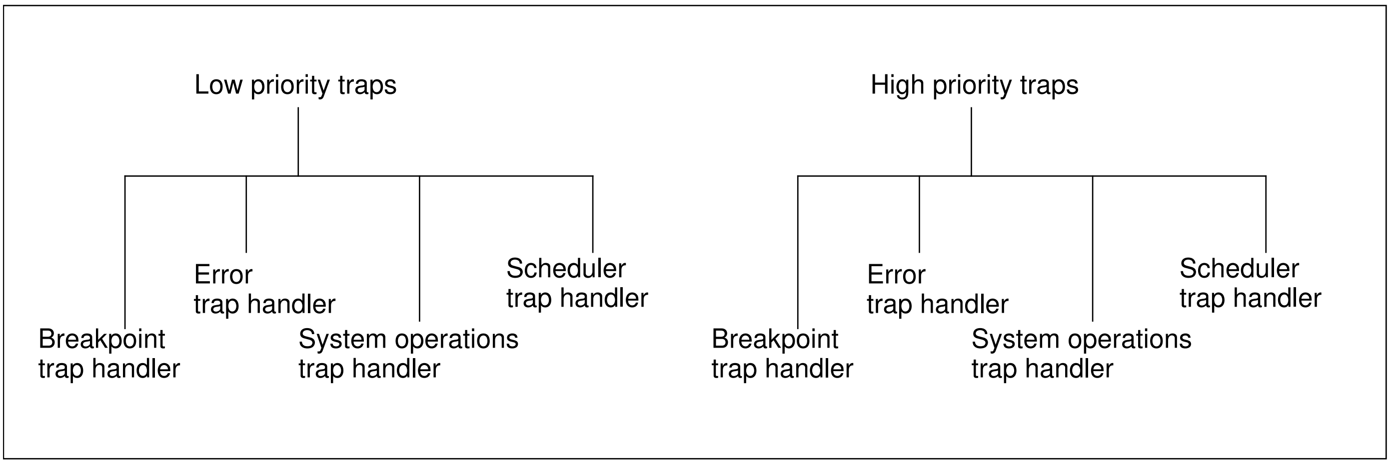


Figure 3.4 Trap arrangement

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 microscheduler rather than by an executing process.

Trap groups encoding is shown in Table 3.3 below. These codes are used to identify trap groups to various instructions.

Trap group	Code
Breakpoint	0
CPU Errors	1
System operations	2
Scheduler	3

Table 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* is executed with an invalid operand.
LoadTrap	4	2	When the trap descriptor is read with the *ldtraph* instruction or when the trapped process status is read with the *ldtrapped* instruction.
StoreTrap	5	2	When the trap descriptor is written with the *sttraph* instruction or when the trapped process status is written with the *sttrapped* instruction.
InternalChannel	6	3	Scheduler trap from internal channel.
ExternalChannel	7	3	Scheduler trap from external channel.
Timer	8	3	Scheduler trap from timer alarm.
Timeslice	9	3	Scheduler trap from timeslice.
Run	10	3	Scheduler trap from *runp* (run process) or *startp* (start process).
Signal	11	3	Scheduler trap from *signal*.
ProcessInterrupt	12	3	Start executing a process at a new priority level.
QueueEmpty	13	3	Caused by no process active at a priority level.
CauseError	15 (Status only)	Any, encoded 0-3	Signals that the *causeerror* instruction set the trap flag.

Table 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. A null **Wptr** indicates that a trap handler has not been installed.	Base + 2
Status	Contains the **Status** register that the trap handler starts with.	Base + 1
Enables	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 3.4 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 **Breg**.

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 ST20450 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 ST20450 interrupt subsystem supports eight prioritized interrupts. This allows nested pre-emptive interrupts for real-time system design.

All interrupts have a higher priority than the high priority process queue. Each interrupt level has a higher priority than the previous (interrupt 7 is the highest priority, interrupt 0 is lowest priority) and each level supports only one software handler process.

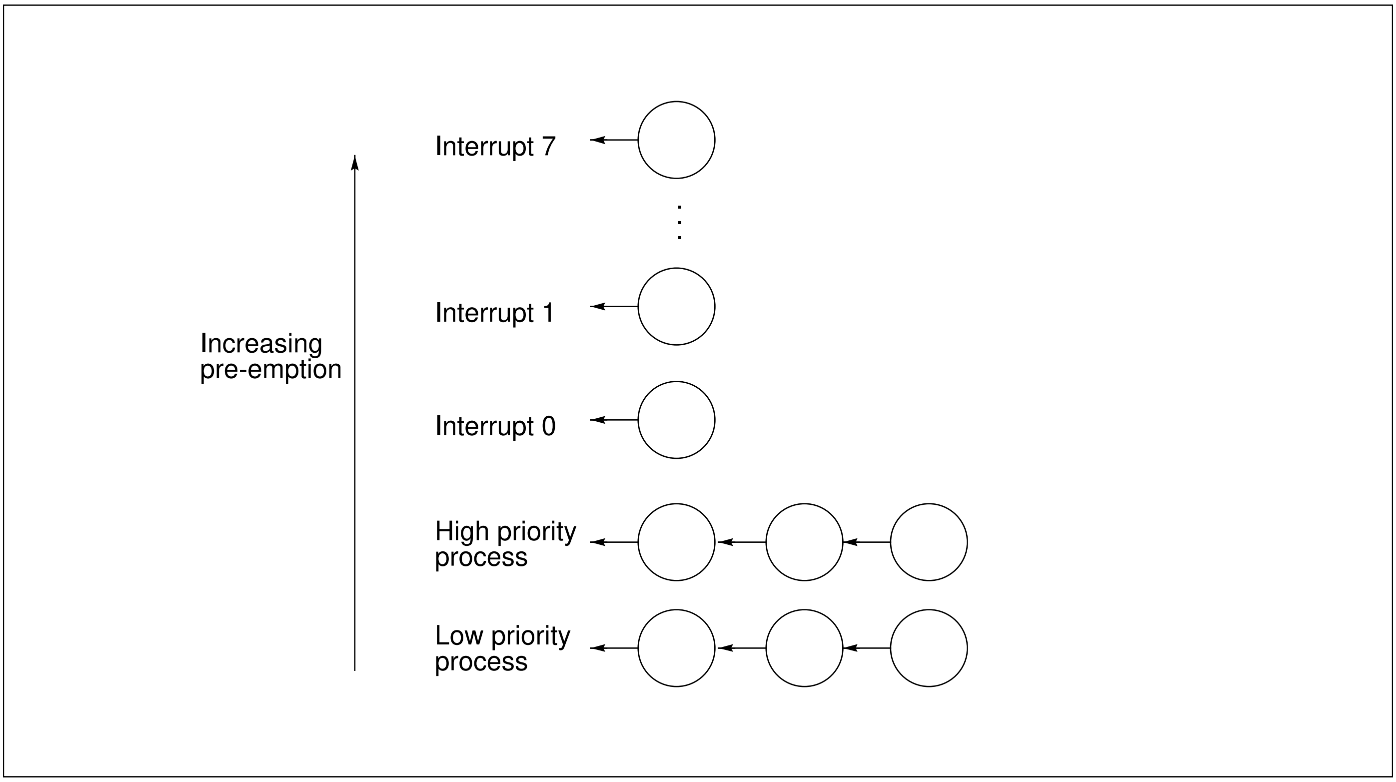


Figure 4.1 Interrupt priority

Interrupts on the ST20450 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 **Interrupt0-7** pin
- software asserting an interrupt in a bit mask

4.1 Interrupt vector table

Each interrupt level has an external **Interrupt** pin to trigger it and a vector table used to associate a software handler with the interrupt level.

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.

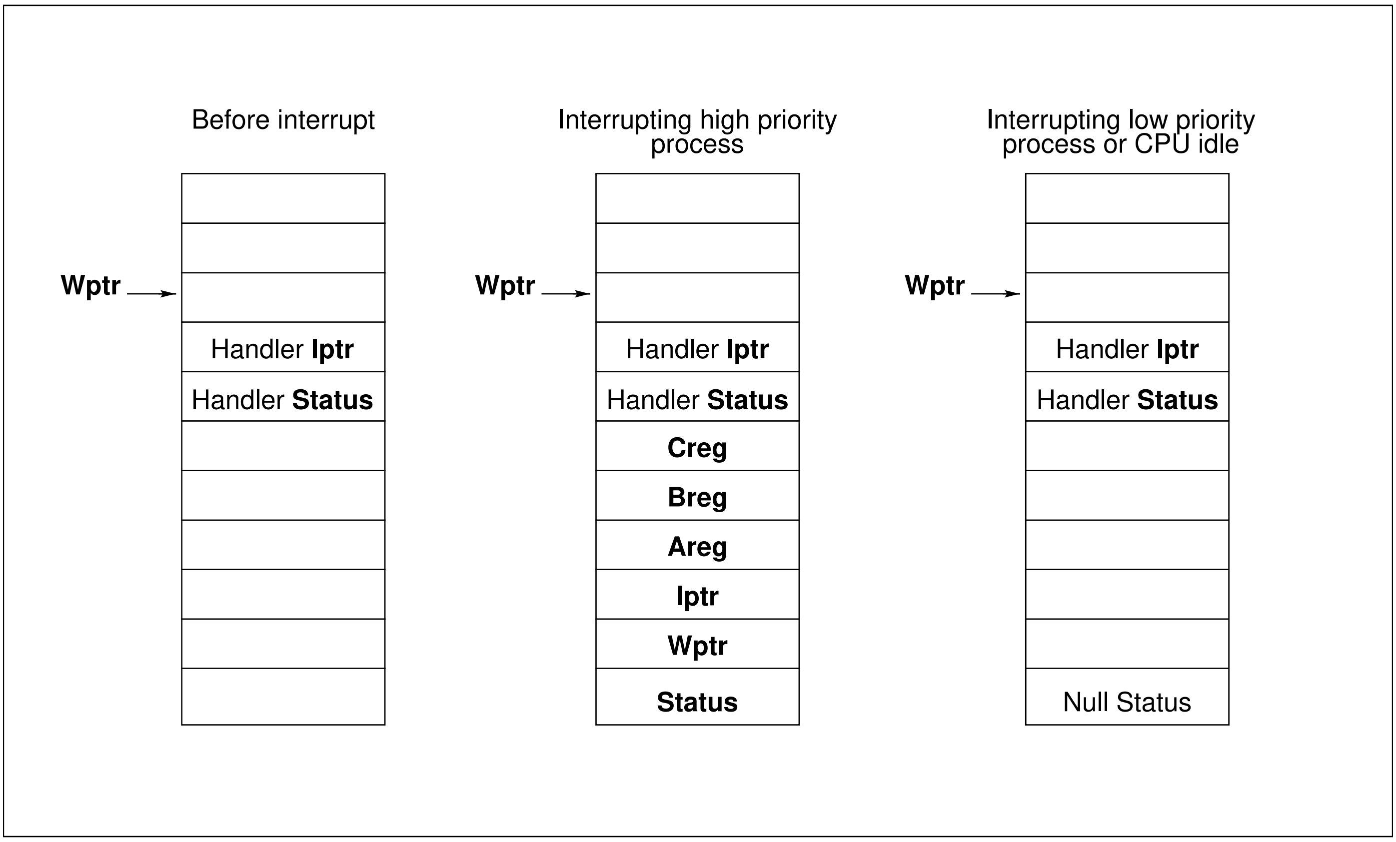


Figure 4.2 State of interrupted process

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, 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 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* and *stshadow* 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

The interrupt latency is dependant 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 Pre-emption and interrupt priority

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

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

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.

4.6.1 HandlerWptr0-7 registers

The **HandlerWptr0-7** registers (1 per interrupt) contain a pointer to the workspace of the interrupt handler.

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.

HandlerWptr0-7	**#20000000 to #2000001C**	**Read/Write**
Bit	**Bit field**	**Function**
31:2	HandlerWptr	Pointer to the workspace of the interrupt handler.
1:0		Reserved. Write 0.

Table 4.1 Bit fields in the **HandlerWptr0-7** registers - one register per interrupt

4.6.2 TriggerMode0-7 registers

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

TriggerMode0-7	#20000040 to #2000005C	Read/Write
Bit	**Bit field**	**Function**
2:0	**Trigger**	Control the triggering condition of the **Interrupt** pin, 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 Bit fields in the **TriggerMode0-7** registers - 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.

4.6.3 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 them to enable them.

Mask	#200000C0	Read/Write
Bit	**Bit field**	**Function**
0	**Interrupt0Enable**	When set to 1, interrupt 0 enabled. When 0, interrupt 0 disabled.
1	**Interrupt1Enable**	When set to 1, interrupt 1 enabled. When 0, interrupt 1 disabled.
2	**Interrupt2Enable**	When set to 1, interrupt 2 enabled. When 0, interrupt 2 disabled.
3	**Interrupt3Enable**	When set to 1, interrupt 3 enabled. When 0, interrupt 3 disabled.
4	**Interrupt4Enable**	When set to 1, interrupt 4 enabled. When 0, interrupt 4 disabled.
5	**Interrupt5Enable**	When set to 1, interrupt 5 enabled. When 0, interrupt 5 disabled.
6	**Interrupt6Enable**	When set to 1, interrupt 6 enabled. When 0, interrupt 6 disabled.
7	**Interrupt7Enable**	When set to 1, interrupt 7 enabled. When 0, interrupt 7 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 Bit fields in the **Mask** register

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

Set-Mask (address #200000C4) 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 #200000C8) 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.

4.6.4 Pending register

The **Pending** register is an 8 bit register with each bit controlled by the corresponding interrupt pin. 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	**#20000080**	**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 Bit fields in the **Pending** register

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

Set-Pending (address #20000084) 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 #20000088) 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.

4.6.5 Exec register

The **Exec** register is an 8 bit register which keeps track of the currently executing and pre-empted 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	**#20000100**	**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 Bit fields in the **Exec** register

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

Set-Exec (address #20000104) 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 #20000108) 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 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 5.1.

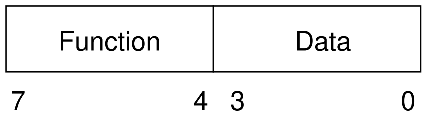


Figure 5.1 Instruction format

For further information on the instruction set refer to the *ST20 Instruction Set Manual (document number 72-TRN-273-00).*

5.1 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.

5.2 Instruction characteristics

The Primary Instructions Table 5.3 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 instruction will be *nfix*. Examples of *pfix* and *nfix* coding are given in Table 5.1.

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 5.1 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 5.2.

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 5.2 Instruction features

5.3 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 5.3 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	lddevid	1	load device identity	
27FE	ldmemstartval	1	load value of **MemStart** address	

Table 5.4 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 5.5 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 5.6 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 5.7 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 5.8 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 5.9 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 5.10 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 5.11 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 5.12 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 5.13 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 5.14 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 5.15 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 5.16 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 5.17 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 5.18 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 5.19 Device access instructions

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

Table 5.20 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 5.21 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 5.22 Trap handler instructions

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

Table 5.23 Processor initialization and no operation instructions

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

Table 5.24 Clock instructions

6 Memory map

The ST20450 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, on-chip peripherals can be accessed via the device access instructions (see Table 5.19).

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.

6.1 System memory use

The ST20450 has a signed address space where the address ranges from **MinInt** (#80000000) at the bottom to **MaxInt** (#7FFFFFFF) at the top. The ST20450 has an area of 16 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 appropriate instructions.

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.

6.1.1 Subsystem channels memory

Each DMA 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 channels

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 one of these channels.

6.1.2 Trap handlers memory

The area of memory reserved for trap handlers is broken down hierarchically. Full details on trap handlers is given in see Section 3.6 on page 16.

- 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).
- 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.

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

6.2 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 of up to 256 bytes down in the ROM program. For large ROM programs it may then be necessary to encode a longer negative jump to reach the start of the routine.

6.3 Internal peripheral space

On-chip peripherals are mapped to addresses in the top half of memory bank 2 (address range #20000000 to #3FFFFFFF). They can only be accessed by the device access instructions (see Table 5.19). When used with addresses in this range, the device instructions access the on-chip peripherals rather than external memory. For all other addresses the device instructions access memory. Standard load/store instructions to these addresses will access external memory.

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
	↑ #20004000	Other on-chip peripherals (registers accessed via CPU device accesses)
	↑ #20003000	Diagnostic controller peripheral (registers accessed via CPU device accesses)
	↑ #20002000	EMI controller peripheral (registers accessed via CPU device accesses)
	↑ #20001000	Low-power controller peripheral (registers accessed via CPU device accesses)
	↑ #20000000	Interrupt controller peripheral (registers accessed via CPU device accesses)
	↑ #00000000	External peripherals
MemStart	↑ #80000140	User code/Data/Stack
	#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

Figure 6.1 ST20450 memory map

	ADDRESS	USE
	#80000050	High priority Breakpoint trapped process
TrapBase	#80000040	High priority Breakpoint trap handler
	#8000003C	RESERVED
	#80000038	
	#80000034	
	#80000030	
	#8000002C	
	#80000028	
	#80000024	
	#80000020	Event channel
	#8000001C	Link3 (boot) input channel
	#80000018	Link2 (boot) input channel
	#80000014	Link1 (boot) input channel
	#80000010	Link0 (boot) input channel
	#8000000C	Link3 output channel
	#80000008	Link2 output channel
	#80000004	Link1 output channel
MinInt	#80000000	Link0 output channel

Figure 6.1 ST20450 memory map

7 Memory subsystem

The memory system consists of SRAM and an external memory interface (EMI). The specific details on the operation of the EMI are described separately in Chapter 8.

7.1 SRAM

There is an internal memory module of 16 Kbytes of SRAM. The internal SRAM is mapped into the base of the memory space from **MinInt** (#80000000) extending upwards, as shown in Figure 7.1.

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

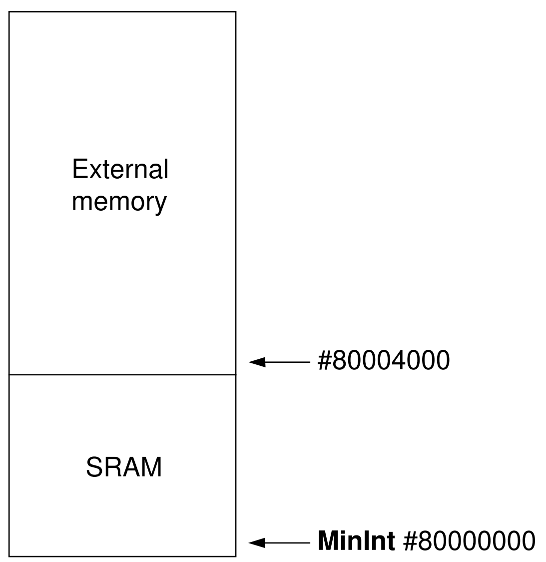


Figure 7.1 SRAM mapping

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

An external control (**DisableRAM**) is provided which can be used to disable internal RAM.

8 External memory interface

The External Memory Interface (EMI) controls the movement of data between the ST20450 and off-chip memory.

The EMI can access a 4 Gbyte physical address space, and provides sustained transfer rates of up to 100 Mbytes/s for SRAM, and up to 89 Mbytes/s using page-mode DRAM. It is designed to support memory subsystems with minimal (often zero) external support logic.

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 8.1). 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.

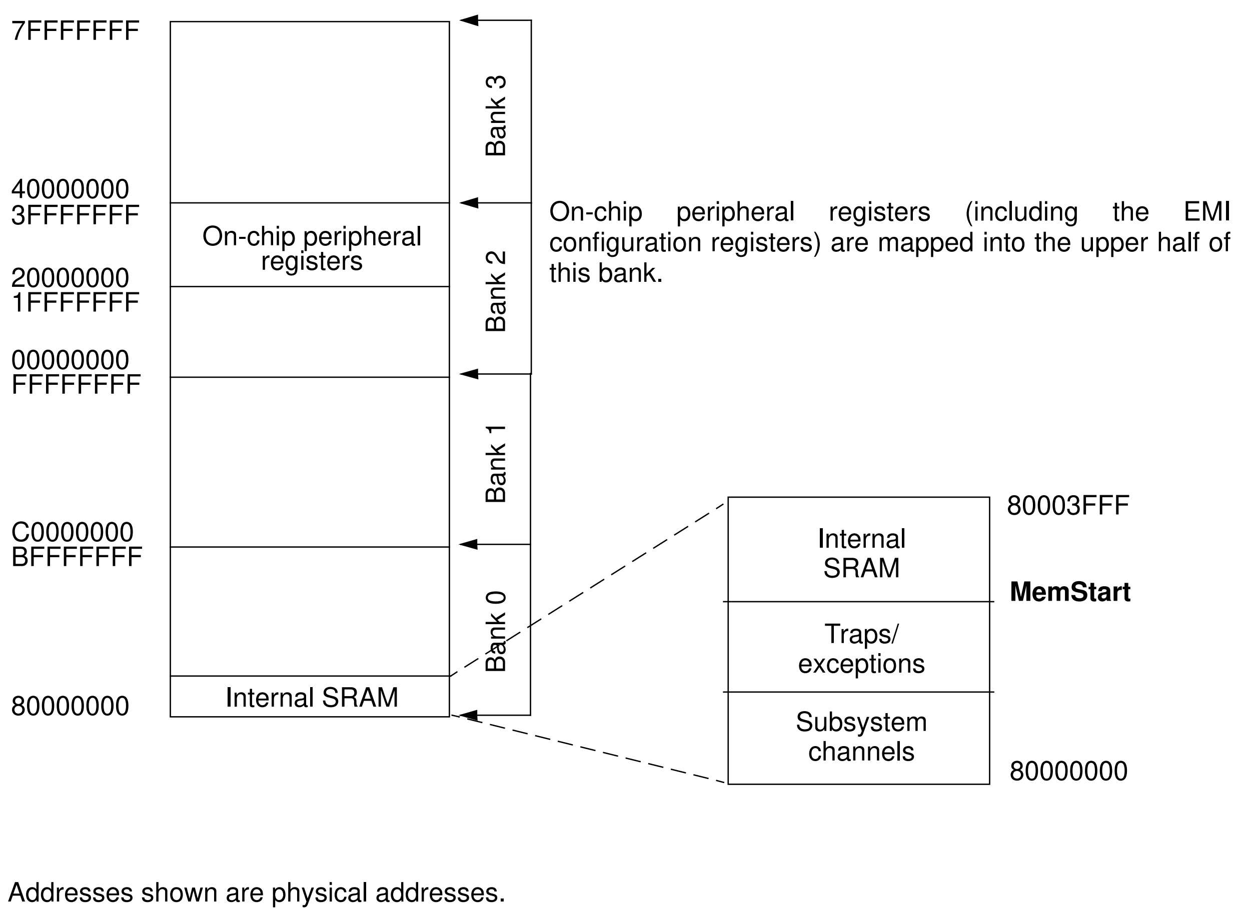


Figure 8.1 Memory allocation

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.

As the banks are of a fixed size and cover the whole address space, range checking of addresses is not possible. This means that software tools must be aware of the physical external memory capacity (this is possible with the configurer). Also, the current software tools cannot utilize discontinuous memory (for example, 32K SRAM in bank 0 and 4M DRAM in bank 1), therefore if mixed memory types must be placed contiguously in the memory map the division must be decoded and handled off-chip.

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.

8.1 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.

MemAddr2-31

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 ST20450 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 **MemAddr2-31** 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 8.1 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 8.1 **notMemBE0-3** pins

notMemRAS0-3

One programmable RAS strobe is allocated to each of the four banks 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

One programmable CAS strobe is allocated to each of the four banks 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. For banks which do not contain DRAM the **notMemCAS** pin is available as a general purpose programmable strobe.

notMemPS0-3

These additional general purpose programmable strobes (one per bank) may be programmed in the same way as the **notMemCAS0-3** strobes.

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 asynchronous **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 8.2 lists the processor pin state while **MemGranted** is asserted.

MemGranted asserted	
Pin name	**Pin state**
MemAddr2-31	floating
MemData0-31	floating
notMemBE0-3	inactive
notMemRAS0-3	inactive
notMemCAS0-3	inactive
notMemPS0-3	inactive
notMemRf	inactive
notMemRd	inactive
MemRefPend	active

Table 8.2 Pin states while **MemGranted** is asserted

MemRefPend

If any of the four banks are configured to contain DRAM, then the **MemRefPend** pin indicates to external logic that the programmed refresh interval is complete and requests the external buses in order to perform a refresh cycle.

The **MemRefPend** signal is held high until external memory is relinquished by the DMA agent. Once **MemReq** has been taken low, only one refresh transaction will be performed, even if several refresh intervals have elapsed between the assertion of **MemRefPend** and the removal of **MemReq**. Refresh transactions will resume once the next refresh interval is complete.

MemRefPend may be configured to signal all pending EMI activity or just pending refresh transactions.

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.

ProcClkOut

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

BootSrce0-1

The **BootSrce0-1** pins determine whether the ST20450 will boot from link or from ROM. When the **BootSrce0-1** pins are both held low the ST20450 will boot from its link. If either or both pins are high the ST20450 will boot from ROM, as shown in Table 8.3. Boot code is run from a slow external ROM placed in bank 3 (at the top of memory). The **BootSrce0-1** pins also encode the size of bank 3. This overrides the value in the configuration registers for the **PortSize** for bank 3.

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 8.3) will be overridden by the value on the **BootSrce0-1** pins.

If the ST20450 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.

BootSrce1:0	Function
0:0	Boot from link. The ST20450 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 8.3 BootSrce0-1 pins

DisableRAM

Internal SRAM can be disabled by setting the **DisableRAM** pin high, enabling systems to operate in external memory only.

When the **DisableRAM** pin is low, i.e. internal SRAM is active, all addresses between **MinInt** and **MinInt+16K** are directed to internal SRAM and external memory at these addresses is never accessed.

8.2 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 8.3).

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 ST20450. The EMI can read bytes, half-words or words. It always reads the size of the bank.

During read or write cycles the ST20450 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 **MemAddr2-31**, however the low order address bits **A0** and **A1** have different functions depending on the size of the external data bus, see Table 8.1. 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 8.2. 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.

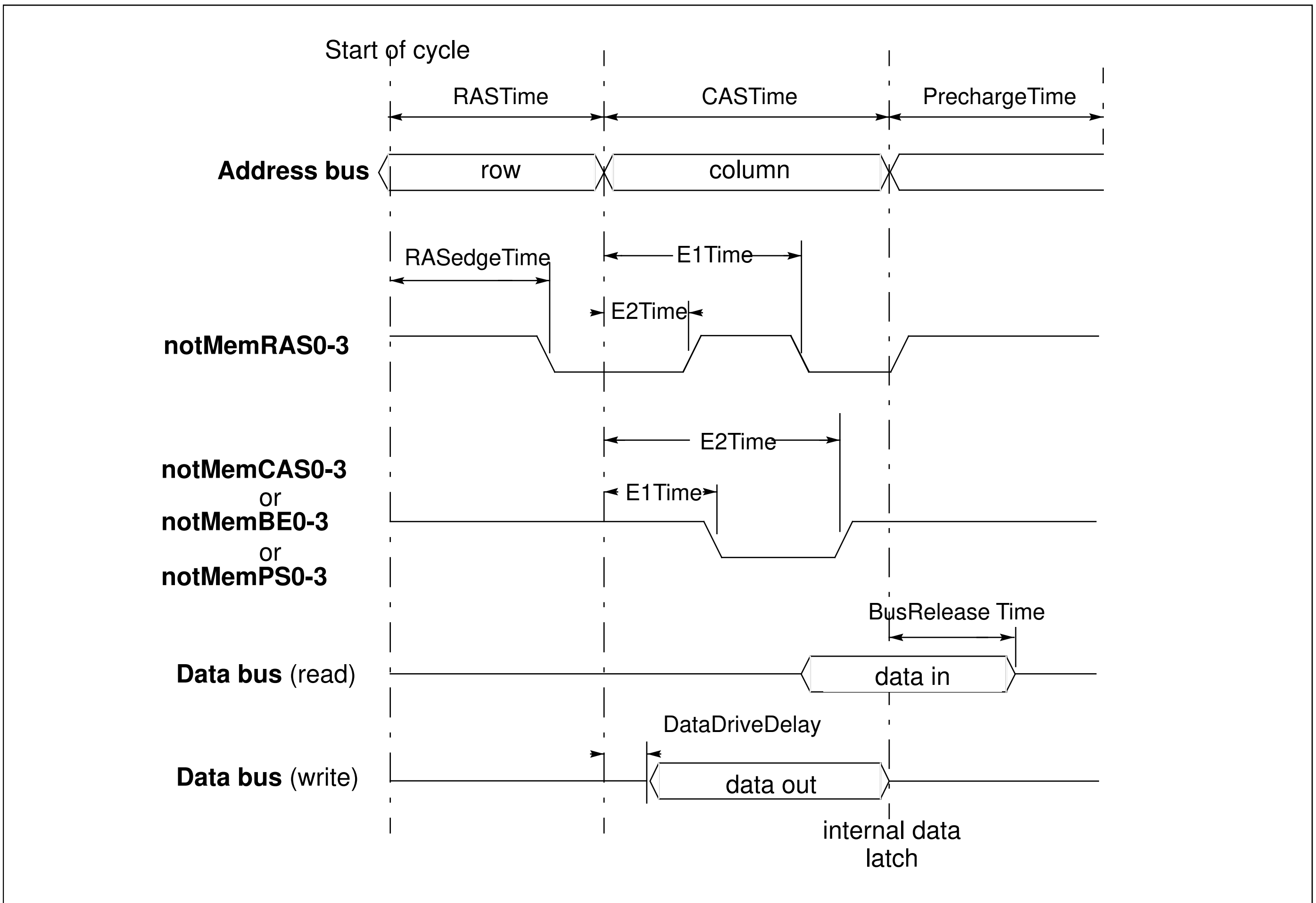


Figure 8.2 Generic memory cycle

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 at the end of **CASTime**.

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.

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

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.

Note, no refreshes take place unless a **DRAMinitialize** command in the **ConfigCommand** register (see Section 8.3.1 on page 51) is performed.

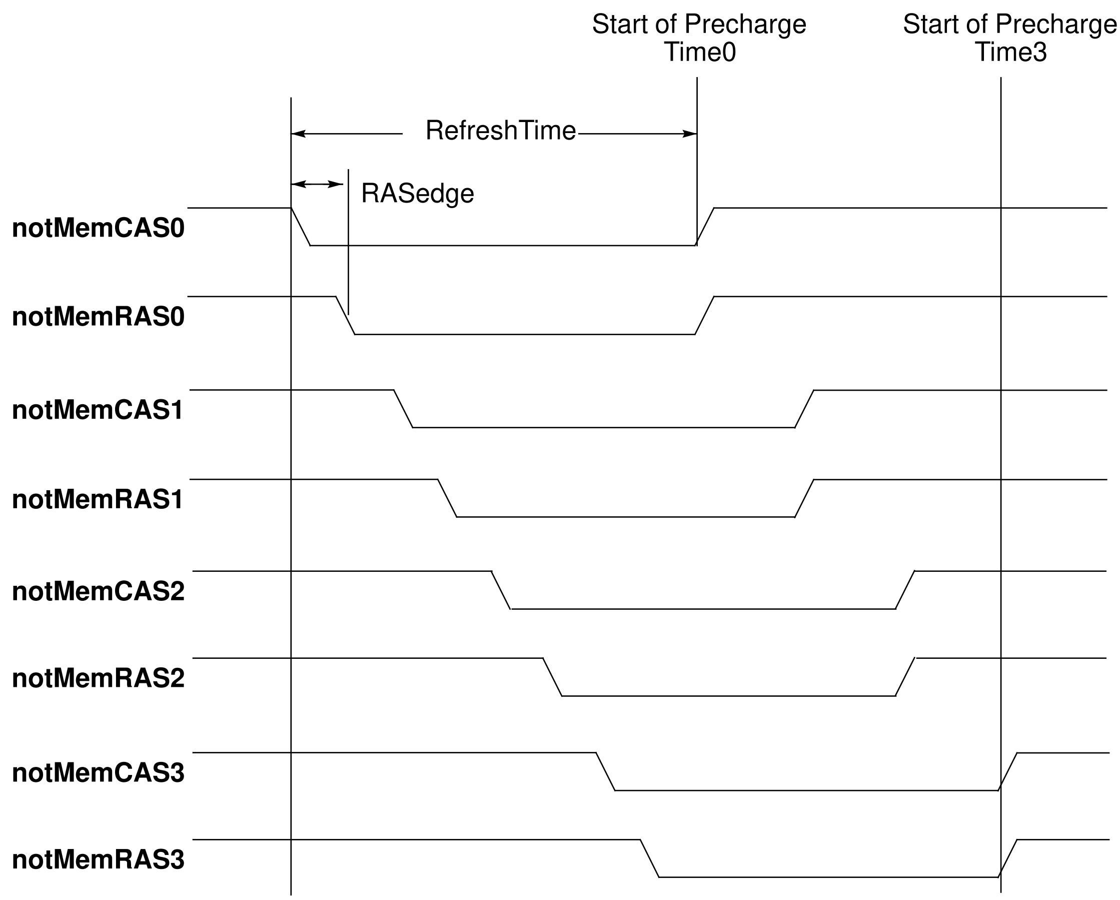


Figure 8.3 Refresh

8.2.2 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 8.4 and Figure 8.5 show the extension of the external memory cycle and the delaying of strobe transitions.

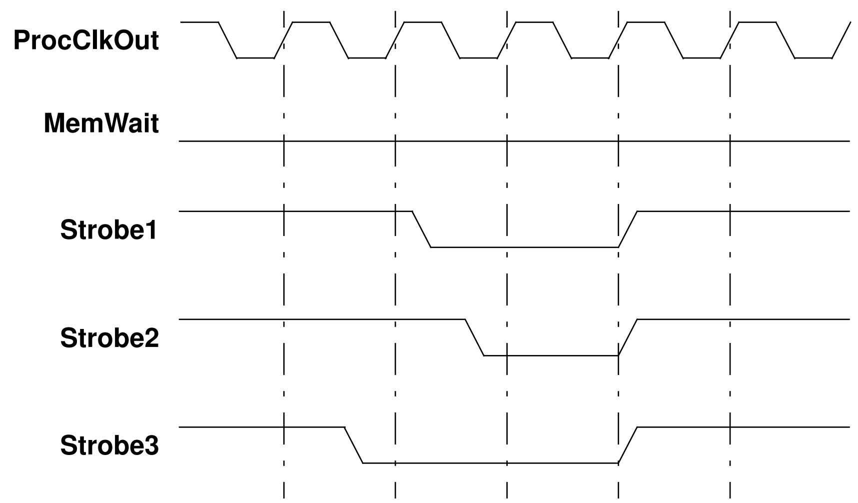


Figure 8.4 Strobe activity without **MemWait**

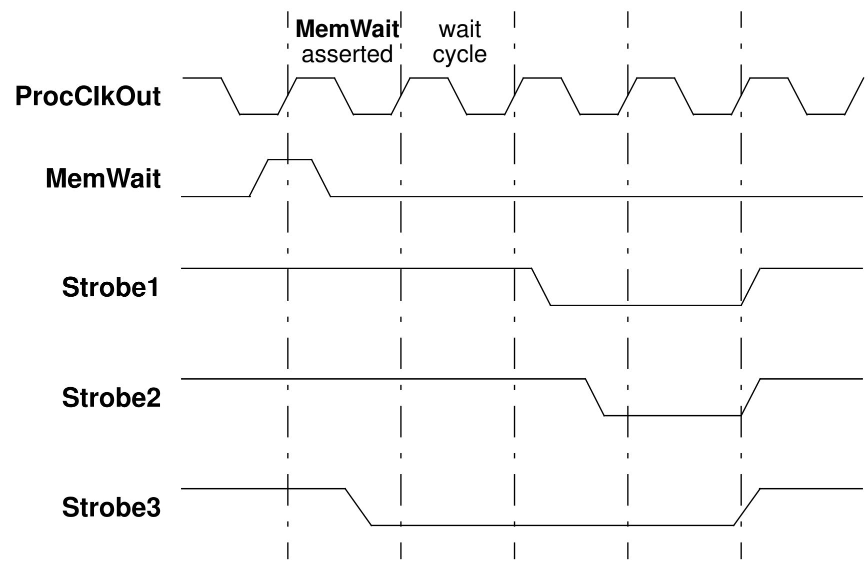


Figure 8.5 Strobe activity with **MemWait**

8.3 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 8.4. 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 ST20450 memory map, see Figure 6.1 on page 39.

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 8.4 EMI configuration register addresses

8.3.1 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 8.5.

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 8.5 **ConfigCommand** register

8.3.2 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 8.6 **ConfigStatus** register

8.3.3 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 8.6.

ConfigDataField0 - banks 0, 1, 2 and 3

Bits	31:2	1	0
Field	RASbits31:2	Page Mode	

ConfigDataField1

bank 0

Bits	31:28	27:24	23:22	21:18	17	16	15:12	11	10	9	8	7	6:2	1:0
Field	CAStime	BusReleaseTime	RAStime	DRAM3:0	CP mask		PrechargeTime	Refr T0	PT =0	RAS t=0	Wait en		ShiftAmount	Port Size

bank 1

Bits	31:28	27:24	23:22	21:17	16	15:12	11	10	9	8	7	6:2	1:0
Field	CAStime	BusReleaseTime	RAStime	RefreshRASedgeTime		PrechargeTime	Refr T1	PT =0	RAS t=0	Wait en		ShiftAmount	Port Size

bank 2

Bits	31:28	27:24	23:22	21:16	15:12	11	10	9	8	7	6:2	1:0
Field	CAStime	BusReleaseTime	RAStime	RefreshInterval5:0	PrechargeTime	Refr T2	PT =0	RAS t=0	Wait en	BR Max0	ShiftAmount	Port Size

bank 3

Bits	31:28	27:24	23:22	21:16	15:12	11	10	9	8	7	6:2	1:0
Field	CAStime	BusReleaseTime	RAStime	RefreshInterval11:6	PrechargeTime	Refr T3	PT =0	RAS t=0	Wait en	BR Max1	ShiftAmount	Port Size

ConfigDataField2 - banks 0, 1, 2 and 3

Bits	31:30	29	28	27:26	25:24	23:22	21	20:19	18	17:15	14	13:12	11	10	9:8	7	6	5:4	3	2	1:0
Field	CASe2 active	CASe 2LSB	CASe 1LSB	CASe1 active	RASedge active	RASe2 active	RASe 2LSB	RASe1 active	RASe 1LSB	RASedgeTime	PSe2 LSB	PSe2 active		PSe1 LSB	PSe1 active	DD 1	BEe2 LSB	BEe2 active	DD 0	BEe1 LSB	BEe1 active

ConfigDataField3 - banks 0, 1, 2 and 3

Bits	31:28	27:24	23:20	19:16	15:12	11:8	7:4	3:0
Field	CASe2timeMSB	CASe1timeMSB	RASe2timeMSB	RASe1timeMSB	PSe2timeMSB	PSe1timeMSB	BEe2timeMSB	BEe1timeMSB

Figure 8.6 **ConfigDataField0-3** registers

8.3.4 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 111111111111111111110000000000

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

RASbits31:2 111111111111111111111000000000

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 8.7 **ConfigDataField0** format for transfers to/from register bank 0

SGS-THOMSON MICROELECTRONICS

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
17	**CyclePendingMask**	Masks the memory access cycle. Determines whether **MemRefPend** indicates pending refresh cycles or pending memory access during DMA.	
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
16, 7		Reserved	

Table 8.8 **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.

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.	
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.	
11		Reserved	

Table 8.9 **ConfigDataField2** format for transfers to/from register bank 0

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 8.10.

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

Table 8.10 Active bit settings

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 8.11 **ConfigDataField3** format for transfers to/from register bank 0

8.3.5 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 8.7, Table 8.9 and Table 8.11 in Section 8.3.4.

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 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	**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 8.12 **ConfigDataField1** format for transfers to/from register bank 1

8.3.6 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.

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

The **ConfigDataField0**, **ConfigDataField2** and **ConfigDataField3** registers have the same format for transfers to/from register bank 2 as those given for transfers to/from register bank 0, see tables 8.7, 8.9 and 8.11 in Section 8.3.4 on page 54.

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 8.14.

ConfigDataField1	EMI base address + #04		Read/Write
Bit	**Bit field**	**Function**	**Units**
1:0	**PortSize**	Bit width of the bank (8/16/32-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.	
7	**BusRelMax0**	This is a 2-bit value (**BusRelMax1** is bit 7 of **ConfigDataField1** for bank 3, refer Table 8.14) 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	**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
15:12	**PrechargeTime**	Duration of precharge time.	Cycles
21:16	**RefreshInterval5:0**	This is a 12-bit value (**RefreshInterval11:6** is bits 21:16 of **ConfigDataField1** for bank 3, refer Table 8.14) 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 8.13 **ConfigDataField1** format for transfers to/from register bank 2

8.3.7 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 8.7, Table 8.9 and Table 8.11 in Section 8.3.4.

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 RAStime. 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 8.13) 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 8.13) 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 8.14 **ConfigDataField1** format for transfers to/from register bank 3

8.3.8 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 (**PadDrive**) 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 8.15.

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

Table 8.15 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 **ProcClkOut** 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 **notMemRAS2**, **notMemCAS2**, **notMemPS2**
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	**A21-24**	Drive strength of pads **MemAddr21-24**
23:22	**A25-31**	Drive strength of pads **MemAddr25-31**
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	**ProcClkEnable**	When 1, **ProcClkOut** pin enabled. When 0 (default state on reset), the **ProcClkOut** pin is disabled, thus reducing power.
30		Reserved

Table 8.16 **ConfigDataField3** format for transfers to/from **PadDrive** register

8.4 EMI initialization

8.4.1 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.

8.4.2 Bootstrap

When external reset is removed, the ST20450 will start to execute bootstrap code from the area of memory determined by the setting of the **BootSrce0-1** pins (see Table 8.3 on page 46).

If the ST20450 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 9.2.3 on page 65) down the link before loading the bootstrap into external memory and executing it.

8.4.3 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 8.18 are also set in the configuration registers.

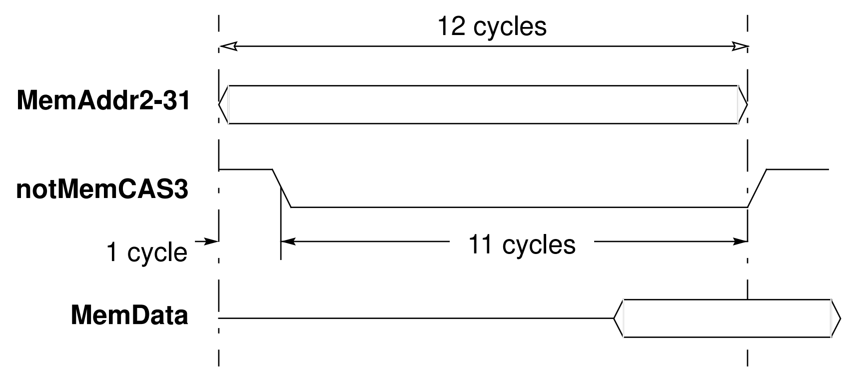


Table 8.17 Timing of default access

Parameter	Default value
RASbits31:2	#0 (all banks)
PageMode	Cleared (all banks)
PortSize	Value on **BootSrce0-1** 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)
CyclePendingMask	Cleared
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
ProcClkEnable	**ProcClkOut** pin disabled

Table 8.18 Default parameters

9 System services

The system services module includes the control system, the PLL, test access port and power control. System services include all the necessary logic to initialize and sustain operation of the device and also includes error handling and analysis facilities.

9.1 Reset, initialization and debug

The ST20450 is controlled by a **notRST** pin which is a global power-on-reset. The CPU can also be controlled by **CPUReset** and **CPUAnalyse** signals.

An additional control signal (**LPIn**) re-starts the ST20450 from low power mode (see Section 11.2 on page 68 for further details on low power operation).

9.1.1 Reset

notRST initializes the device and causes it to enter its boot sequence which can either be in off-chip ROM or can be received down a link (see Section 9.2 on bootstrap). **notRST** must be asserted at power-on.

When **notRST** is asserted low, all modules are forced into their power-on reset condition. The clocks are stopped. The rising edge of **notRST** is internally synchronized and delayed until the clocks are stable before starting the initialization sequence.

CPUReset is provided as a functional reset which is quicker to reboot as the PLL is not reset. In other respects the effect is the same as **notRST**. **CPUReset** can be used in conjunction with **CPUAnalyse**.

The **ResetRespOut** signal provides additional information with regard to the progress of internal reset activities and generally as a CPU running signal. The **ResetRespOut** signal when low signals that the CPU has halted following a halt-on-error or a **CPUAnalyse**.

9.1.2 CPUAnalyse

If **CPUAnalyse** is taken high when the ST20450 is running, the ST20450 will halt at the next descheduling point. **CPUReset** may then be asserted. When **CPUReset** comes low again the ST20450 will be in its reset state, and information on the state of the machine when it was halted by the assertion of **CPUAnalyse**, is maintained permitting analysis of the halted machine.

An input link will continue with outstanding transfers. An output link will not make another access to memory for data but will transmit only those bytes already in the link buffer. Providing there is no delay in link acknowledgement, the link will be inactive within a few microseconds of the ST20450 halting.

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

9.1.3 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 ST20450 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 ST20450 can alternatively be set to halt-on-error to prevent further corruption and allow postmortem debugging. The ST20450 also supports user defined trap handlers, see Section 4.6 on page 21 for details.

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

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

9.2 Bootstrap

The ST20450 can be bootstrapped from external ROM, internal ROM or from a link. This is determined by the setting of the **BootSrce0-1** pins, see Table 9.1 on page 51. If both **BootSrce0-1** 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 ST20450, before the first instruction is executed after reset.

9.2.1 Booting from ROM

When booting from ROM, the ST20450 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.

9.2.2 Booting from link

When booting from a link, the ST20450 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 ST20450 will start executing code at **MemStart**. The memory space immediately above the loaded code is used as work space. A byte arriving on other links after the control byte has been received, and on the bootstrapping link after the last bootstrap byte, is retained and no acknowledge is sent until a process inputs from the link.

9.2.3 Peek and poke

Any location in internal or external memory can be interrogated and altered when the ST20450 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.

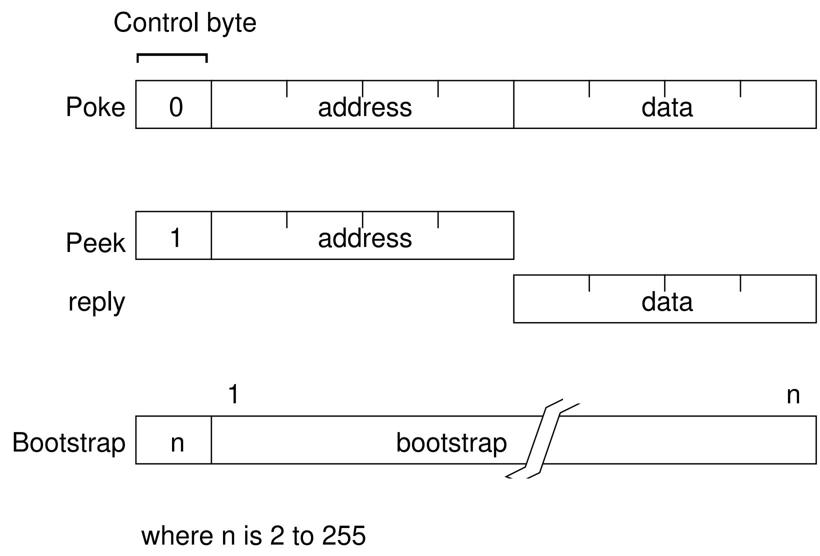


Figure 9.1 Peek, poke and bootstrap

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 ST20450 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 ST20450 will commence reading its bootstrap program. Any link can be used but addresses and data must be transmitted via the same link as the control byte.

10 Test Access Port

The ST20450A conforms to IEEE standard 1149.1, with the exception of the **EventWaiting** pin, as reported in the ST20450 bug list.

The Test Access Port (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 10.1. All other instruction codes are reserved.

<table>
<tr><th colspan="5">Instruction code [a]</th><th>Instruction</th><th>Selected register</th></tr>
<tr><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>EXTEST</td><td>Boundary-Scan</td></tr>
<tr><td>0</td><td>0</td><td>0</td><td>0</td><td>1</td><td>IDCODE</td><td>Identification</td></tr>
<tr><td>0</td><td>0</td><td>0</td><td>1</td><td>0</td><td>SAMPLE/PRELOAD</td><td>Boundary-Scan</td></tr>
<tr><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td><td>BYPASS</td><td>Bypass</td></tr>
</table>

Table 10.1 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, see Appendix A on page 99.

The identification code is 05000011, see Table 10.2.

<table>
<tr><td colspan="8">bit 31</td><td colspan="8"></td><td colspan="8"></td><td colspan="8">bit 0 [a]</td></tr>
<tr><td colspan="4">Mask Rev</td><td>b</td><td colspan="5">ST20 family</td><td colspan="10">Variant</td><td colspan="11">SGS-THOMSON manufacturers id</td><td>c</td></tr>
<tr><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>1</td><td>0</td><td>1</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>1</td><td>0</td><td>0</td><td>0</td><td>1</td></tr>
<tr><td colspan="4">0</td><td colspan="4">5</td><td colspan="4">0</td><td colspan="4">0</td><td colspan="4">0</td><td colspan="4">0</td><td colspan="4">1</td><td colspan="4">1</td></tr>
</table>

Table 10.2 Identification code

a. Closest to **TDO.**
b. 0 indicates SGS-THOMSON part, 1 indicates customer part.
c. Defined as 1 in IEEE 1149.1 standard.

11 Clocks and low power operation

11.1 Clocks

Two on-chip phase locked loops (PLL) generate all the internal high frequency clocks. The PLLs are used to generate the internal clock frequencies needed for the CPU and the OS-Links. Alternatively a direct clock input can provide the system clocks.

Internal clocks may be turned off (including PLLs) enabling power down mode.

There are two clock inputs, one for the system clock (**ClockIn**) and one for the link clock (**LinkClockIn**). **ClockIn** must be in the range 3.2 to 10 MHz for PLL operation, see Table 11.1. **LinkClockIn** must be 5 MHz during PLL operation for compatibility with the Development System and other standard OS-Link products.

The ST20450 can be set to operate in **TimesOneMode**, which is when the PLL is bypassed. During **TimesOneMode** for either clock the relevant input clock must be in the range 0 to 32 MHz and should be nominally 50/50 mark space ratio.

11.1.1 Processor speed select

The speed of the internal processor clock is variable in discrete steps. The clock rate at which the ST20450 runs at is determined by the logic levels applied on the three speed select lines **ProcSpeed0-2** as detailed in Table 11.1. The frequency of **ClockIn** for the processor speeds given in the table is nominally 5 MHz.

ProcSpeed2	ProcSpeed1	ProcSpeed0	Processor clock speed MHz	Processor cycle time ns	Phase lock loop factor (PLLx)	Allowable ClockIn range MHz
0	0	0	TimesOneMode			0 - 32
0	0	1	RESERVED			RESERVED
0	1	0	RESERVED			RESERVED
0	1	1	30	33.3	6	5 - 8.3
1	0	0	40	25.0	8	4 - 6.25
1	0	1	RESERVED			RESERVED
1	1	0	RESERVED			RESERVED
1	1	1	RESERVED			RESERVED

Table 11.1 Processor speed selection

11.2 Low power control

The ST20450 is designed for 0.5 micron, 3.3V CMOS technology and runs at speeds of up to 40 MHz. 3.3V operation provides reduced power consumption internally and allows the use of low power peripherals. In addition, to enhance the potential for battery operation further, a low power power-down mode is available on the ST20450.

The different power levels of the ST20450 are listed below.

- Operating power - power consumed during functional operation.

- Standby power - power consumed during little or no activity. The CPU is idle but ready to immediately respond to an interrupt/reschedule.
- Power-down - internal clocks are stopped and power consumption is significantly reduced. Functional operation is stalled. Normal functional operation can be resumed from previous state as soon as the clocks are stable as all internal logic is static no information is lost during power-down.

  The ST20450 enters power-down when:

  - **LPIn** is high - irrespective of all other device activity.

  The ST20450 exits power-down when:

  - **LPIn** goes low.

In power-down mode the processor and all peripherals are stopped, including the external memory controller and optionally the PLLs. Effectively the internal clock is stopped and functional operation is stalled. On restart the clock is restarted and the chip resumes normal functional operation.

Low power operation can be achieved in one of two ways, as listed below.

- Availability of direct clock input - this allows external control of clocking directly and thus direct control of power consumption. (Available in TimesOneMode only).
- Global system clock may be stopped, using the **LPIn** pin as described above. In this case the external clock remains running. This mechanism allows any PLLs to be kept running (if desired) so that wake-up from low power mode will be fast.

Note, DRAM refresh is not supported in power-down mode and DMA Request requires special attention to work correctly.

11.2.1 Low power configuration registers

The low power controller is allocated a 4k block of memory in the internal peripheral address space. Information on low power mode 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.

LPSysPll

The **LPSysPll** register controls what happens to the System Clock in PLL operation when low power mode is entered.

LPSysPll	**#20001420**	**Read/Write**
Bit	**Bit field**	**Function**
1:0	**LPSysPll**	Determines the system clock PLL when low power mode is entered, as follows: **LPSysPll1:0** — **System clock** 00 — PLL off 01 — PLL reference on 10 — PLL reference on 11 — PLL on

Table 11.2 Bit fields in the **LPSysPll** register

LPLinkPll

The **LPLinkPll** register controls what happens to the Link Clock PLL when low power mode is entered.

LPLinkPll	#20001424	Read/Write
Bit	**Bit field**	**Function**
2:0	**LPLinkPll**	Determines the link clock PLL when low power mode is entered, as follows: **LPLinkPll2:0** **Link clock** 000 PLL off 001 PLL reference on 010 PLL reference on 011 PLL on 100 RESERVED 101 RESERVED 110 RESERVED 111 PLL running and OS-Link inputs clocking during low power.

Table 11.3 Bit fields in the **LPLinkPll** register

SysRatio

The **SysRatio** register is a read only register and gives the speed that the system PLL is running at. It contains the relevant PLL multiply ratio when using a PLL, or contains the value ‘1’ when in **TimesOneMode** for that PLL.

LinkRatio

The **LinkRatio** register is a read only register and gives the speed that the link PLL is running at. It contains the relevant PLL multiply ratio when using a PLL, or contains the value ‘1’ when in **TimesOneMode** for that PLL.

Note that for the OS-Links the clock speed is 2.5 times the data rate, thus a multiply ratio of 10 with a 5 MHz input clock rate is equivalent to a link speed of 20 Mbits/s.

11.3 Wakeup times and power consumption during standby

In standby the system and link PLLs have a number of possible states, determined by the setting of the **LPSysPll** and **LPLinkPll** registers, which allow a compromise between wakeup time and power consumption during standby.

System PLL state during standby	Approximate wakeup time	Power in standby	Notes
Running	3 LPC clocks + 6 system (CPU) clocks	20 mW	
Standby	0.5 ms	2 mW + leakage	1
Off (default)	2 ms	leakage	1

Notes

1 Leakage is 3.5 mW maximum at 125 °C die temperature.

Table 11.4 System PLL options

Link PLL state during standby	Approximate wakeup time	Power in standby	Notes
Running and clocking OS-Link engine	0	40 mW	
Running	3 LPC clocks + 6 system (CPU) clocks	20 mW	
Standby	0.5 ms	2 mW + leakage	1
Off (default)	2 ms	leakage	1

Notes

1 Leakage is 3.5 mW maximum at 125 °C die temperature.

Table 11.5 Link PLL options

11.4 Clocking sources

The low power timer and alarm must be clocked at all times by the following clocking source:

- External clock input (**LPClockIn**) - this clock can be at any rate below 5 MHz provided it is not more than one eighth of the system clock rate.

12 Serial link interface (OS-Link)

The ST20450 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.
- Connection to external peripherals - interface devices are available that allow OS-Links to be interfaced to standard peripherals and buses. A macrocell is also available to give OS-Link to parallel interface conversion for use on external ASIC based peripherals.
- Multiprocessing - OS-Links allow the ST20450 to be directly connected in a multiprocessor system with other OS-Link devices. Inter-processor communication is directly supported in software.

12.1 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 12.1). 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.

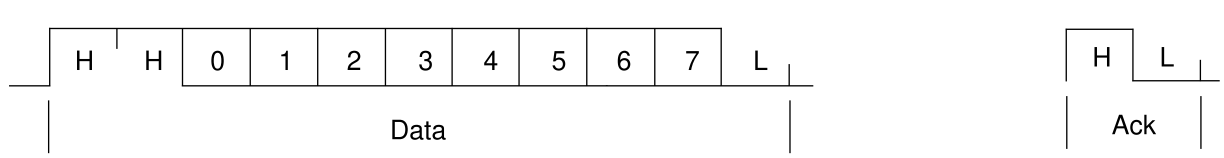


Figure 12.1 OS-Link data and acknowledge formats

12.2 OS-Link speed

The bandwidth of each link is selectable to 10 or 20 Mbits/s, **Link0** can also be set to run at 5 Mbits/s if the **Link0Special** pin is set high, see Table 12.1. Links are not synchronized with the input clock or the processor clock and are insensitive to their phases. Thus links from independently clocked systems may communicate, providing only that the clocks are nominally identical and within specification.

LinkSpeed1	LinkSpeed0	Link0Special	Link1-3 speed	Link0 speed
0	0	X	TimesOneMode	TimesOneMode
0	1	0	10 Mbits/s	10 Mbits/s
1	0	0	20 Mbits/s	20 Mbits/s
1	1	X	RESERVED	RESERVED
0	1	1	10 Mbits/s	5 Mbits/s
1	0	1	20 Mbits/s	10 Mbits/s

Table 12.1 Link speed settings

The links can be run at **TimesOneMode**, i.e. with the PLL bypassed. When the links are in **TimesOneMode** they actually run at the speed of the link clock divided by 2.5. For example, with the link clock at 50 MHz, **Link1-3** run at 20 Mbits/s and **Link0** runs at 10 or 20 Mbits/s depending on the setting of the **Link0Special** pin, see Table 12.2.

LinkSpeed1:0	Link0Special	Link1-3 speed	Link0 speed
00	0	20 Mbits/s	20 Mbits/s
00	1	20 Mbits/s	10 Mbits/s

Table 12.2 Link speeds when set to **TimesOneMode** with the link clock at 50 MHz

12.3 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 12.3. When this is done the line delay should be 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 12.4. 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.

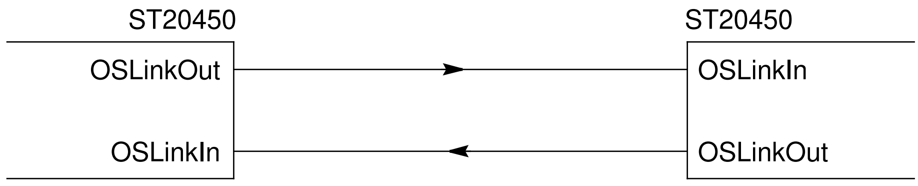


Figure 12.2 OS-Links directly connected

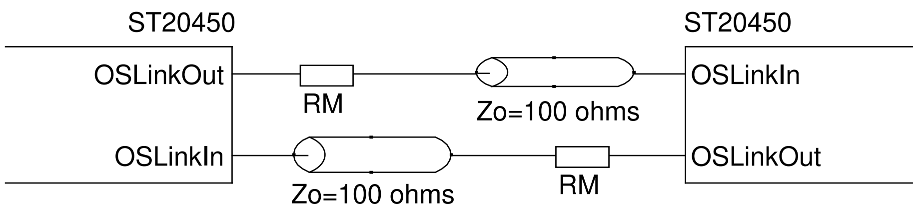


Figure 12.3 OS-Links connected by transmission line

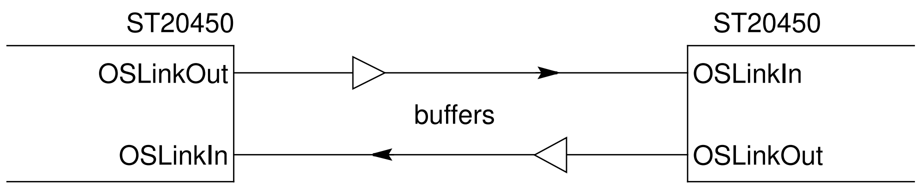


Figure 12.4 OS-Links connected by buffers

12.4 Event

EventReq and **EventAck** provide an asynchronous handshake interface between an external event and an internal process. Event channels provide process synchronization but cannot transfer any data. When an external event takes **EventReq** high the external event channel (additional to the external link channels) is made ready to communicate with a process. When both the event channel and the process are ready the processor takes **EventAck** high and the process, if waiting, is scheduled. **EventAck** is removed after **EventReq** goes low.

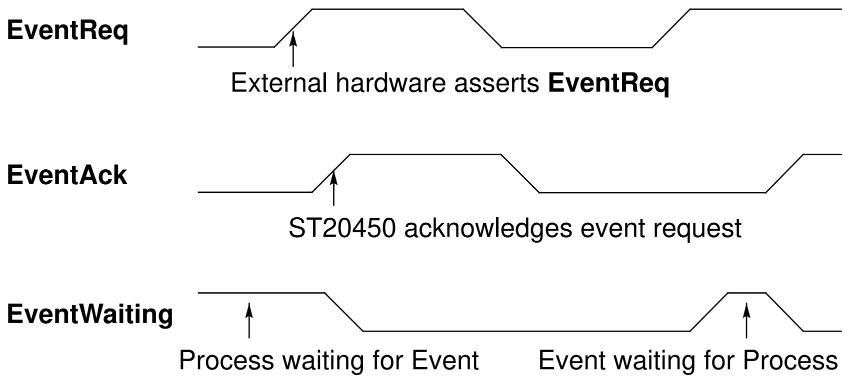


Figure 12.5 Event

EventWaiting is asserted high when a process executes an input on the event channel. Note, the **EventWaiting** pin can only be asserted by executing an *in* instruction, it is not asserted high when an enable channel (*enbc*) instruction is executed on the event channel. **EventWaiting** remains high whilst the device is waiting for or servicing **EventReq** and is returned low when **EventAck** goes high. The **EventWaiting** pin changes near the falling edge of **ProcClockOut** and can therefore be sampled by the rising edge of **ProcClockOut**.

EventWaiting allows a process to control external logic; for example, to clock a number of inputs into a memory mapped data latch so that the event request type can be determined.

Only one process may use the event channel at any given time. If no process requires an event to occur **EventAck** will never be taken high. Although **EventReq** triggers the channel on a transition from low to high, it must not be removed before **EventAck** is high. **EventReq** should be low during **Reset**; if not it will be ignored until it has gone low and returned high. **EventAck** is taken low when **Reset** occurs.

If the process is a high priority one and no other high priority process is running, typical latency is 19 processor cycles, and maximum latency (assuming all memory accesses are internal) is 58 processor cycles.

Setting a high priority task to wait for an event input allows the user to interrupt a program running at low priority. The time taken from asserting **EventReq** to the execution of the microcode interrupt handler in the CPU is four cycles. The following functions take place during the four cycles:

Cycle 1 Sample **EventReq** at pad on the rising edge of **ProcClockOut** and synchronize.

Cycle 2 Edge detect the synchronized **EventReq** and form the interrupt request.

Cycle 3 Sample interrupt vector for microcode ROM in the CPU.

Cycle 4 Execute the interrupt routine for the event rather than the next instruction.

13 Software development

Software development support for the ST20450 is provided by the ST20 Toolset which includes a range of advanced debugging tools supporting a windows based graphical user interface (GUI) for PC and UNIX machines.

13.1 ST20 toolset

The ST20 toolset provides a range of tools to support the developer including:

- Compiler - full ANSI validated optimizing C compiler, with support for assembly language programming.
- Runtime libraries - full ANSI validated libraries.
- Compacting linker and Librarian - support for user, application and run-time libraries.
- Configurer - supports the mapping of an application onto an ST20 processor.
- Mapping tool - conversion of symbolic to absolute addresses.
- ROM tool - support EPROM programming.
- Memory configuration tool - support for initializing the EMI.

Refer to the *ST20 ANSI C Toolset Datasheet (document number 42 1669 00)* for further details.

13.1.1 Debugging and profiling software

Debugging and profiling software includes an extended range of tools supporting a window based GUI appropriate to the development host:

- Windows 3.1 for PC machines
- OSF Motif for UNIX machines
- Profiler - conventional profiling tool.
- Load monitor - profile of the CPU usage, hot/cold spots, computation loading, communications loading.
- INQUEST windowing debugger - providing source and assembler debugging with watchpoints, breakpoints, single stepping, stack trace and symbolic data inspection.

Further details can be found in the *PC ST20 Inquest Datasheet (document number 42 1668 00)* and the *Sun 4 ST20 Inquest Datasheet (document number 42 1667 00)*.

14 Configuration register addresses

This chapter lists all the ST20450 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 can not be accessed using memory instructions.

Register	Address	Size	Set	Clear	Read/Write	Ref page
HandlerWptr0	#20000000	30			R/W	24
HandlerWptr1	#20000004	30			R/W	
HandlerWptr2	#20000008	30			R/W	
HandlerWptr3	#2000000C	30			R/W	
HandlerWptr4	#20000010	30			R/W	
HandlerWptr5	#20000014	30			R/W	
HandlerWptr6	#20000018	30			R/W	
HandlerWptr7	#2000001C	30			R/W	
TriggerMode0	#20000040	3			R/W	24
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	26
Set-Pending	#20000084				W	
Clear-Pending	#20000088				W	
Mask	#200000C0	9			R/W	25
Set-Mask	#200000C4				W	
Clear-Mask	#200000C8				W	
Exec	#20000100	8	Interrupt valid	Interrupt done	R/W	27
Set-Exec	#20000104				W	
Clear-Exec	#20000108				W	
LPSysPll	#20001420	2			R/W	69
LPLinkPll	#20001424	3			R/W	70
SysRatio	#20001500	6			R	70

Table 14.1 ST20450 register addresses

Register	Address	Size	Set	Clear	Read/Write	Ref page
LinkRatio	#20001504	6			R	70
ConfigDataField0	#20002000	32			R/W	52
ConfigDataField1	#20002004	32			R/W	
ConfigDataField2	#20002008	32			R/W	
ConfigDataField3	#2000200C	32			R/W	
ConfigCommand	#20002010	32			W	51
ConfigStatus	#20002020	32			R	52

Table 14.1 ST20450 register addresses

15 Electrical specifications

15.1 Absolute maximum ratings

Symbol	Parameter	Min	Max	Units	Notes
VDD	DC supply voltage		3.6	V	
V_I	Voltage on input pins	GND-0.6	5.75	V	
V_O	Voltage on bi-directional and output pins	GND-0.6	VDD+0.6	V	
I_O	DC output current		25	mA	
T_S	Storage temperature (ambient)	-55	125	°C	
T_A	Temperature under bias (ambient)	-55	125	°C	
PDmax	Power dissipation		1.98	W	

Notes

1 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.

Table 15.1 Absolute maximum ratings

15.2 Operating conditions

Symbol	Parameter	Min	Max	Units	Notes
V_I, V_O	Input or output voltage	0	VDD	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	
T_A	Operating temperature (ambient) ST20450-S	0	70	°C	

Notes

1 Excursions beyond the supplies are permitted but not recommended; see DC characteristics.

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

Table 15.2 Operating conditions

15.3 DC specifications

Symbol	Parameter	Min	Typical	Max	Units	Notes
V_{DD}	Positive supply voltage	3.0	3.3	3.6	V	
V_{IH}	Input logic 1 voltage (bi-directional pins and **LPClockIn** pin)	2.0		VDD+0.5	V	
	Input logic 1 voltage (input pins)	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	1
V_{OHDC}	Output logic 1 voltage	2.4			V	2
V_{OLDC}	Output logic 0 voltage			0.4	V	2
C_{IN}	Input capacitance (input pins)			7	pF	
	Input capacitance (bi-directional pins)			14	pF	
C_{OUT}	Output capacitance			15	pF	

Notes

1 $0 \le V_I \le V_{DD}$

2 I_{load}=2mA

Table 15.3 DC specifications

16 Timing specifications

16.1 EMI timings

The timings are based on the following loading conditions: 50 pF load with the pad drive strengths (see Table 8.15 on page 61 for details on the pad drive strength) as follows:

MemAddr2-31 drive strength at level 1

Strobe drive strength at level 1

MemData0-31 drive strength at level 3

Symbol	No.	Parameter	Min	Max	Units	Notes
tCHAV	1	Reference Clock high to Address valid	-6.8	0.0	ns	
tCLSV	2	Reference Clock low to Strobe valid	-6.0	2.3	ns	
tCHSV	3	Reference Clock high to Strobe valid	-5.7	0.0	ns	
tRDVCH	4	Read Data valid to Reference Clock high		12.8	ns	
tCHRDX	5	Read Data hold after Reference Clock high		-6.5	ns	
tCLWDV	6	Reference Clock low to Write Data valid	-7.3	2.9	ns	
tCHWDV	7	Reference Clock high to Write Data valid	-6.3	1.9	ns	
tCHRSV	8	Reference Clock high to remaining Strobes valid	-4.9	5.6	ns	
tCHPH	9	Reference Clock high to ProcClkOut	-6.1	0.7	ns	
tWVCH	10	MemWait valid to Reference Clock high		12.4	ns	
tCHWX	11	MemWait hold after Reference Clock high		-6.5	ns	
tRVCH	12	MemReq valid to Reference Clock high		12.4	ns	
tCHRX	13	MemReq hold after Reference Clock high		-6.5	ns	

Table 16.1 EMI cycle timings

Note, 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 dependant 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 a threshold of 1.5 V.

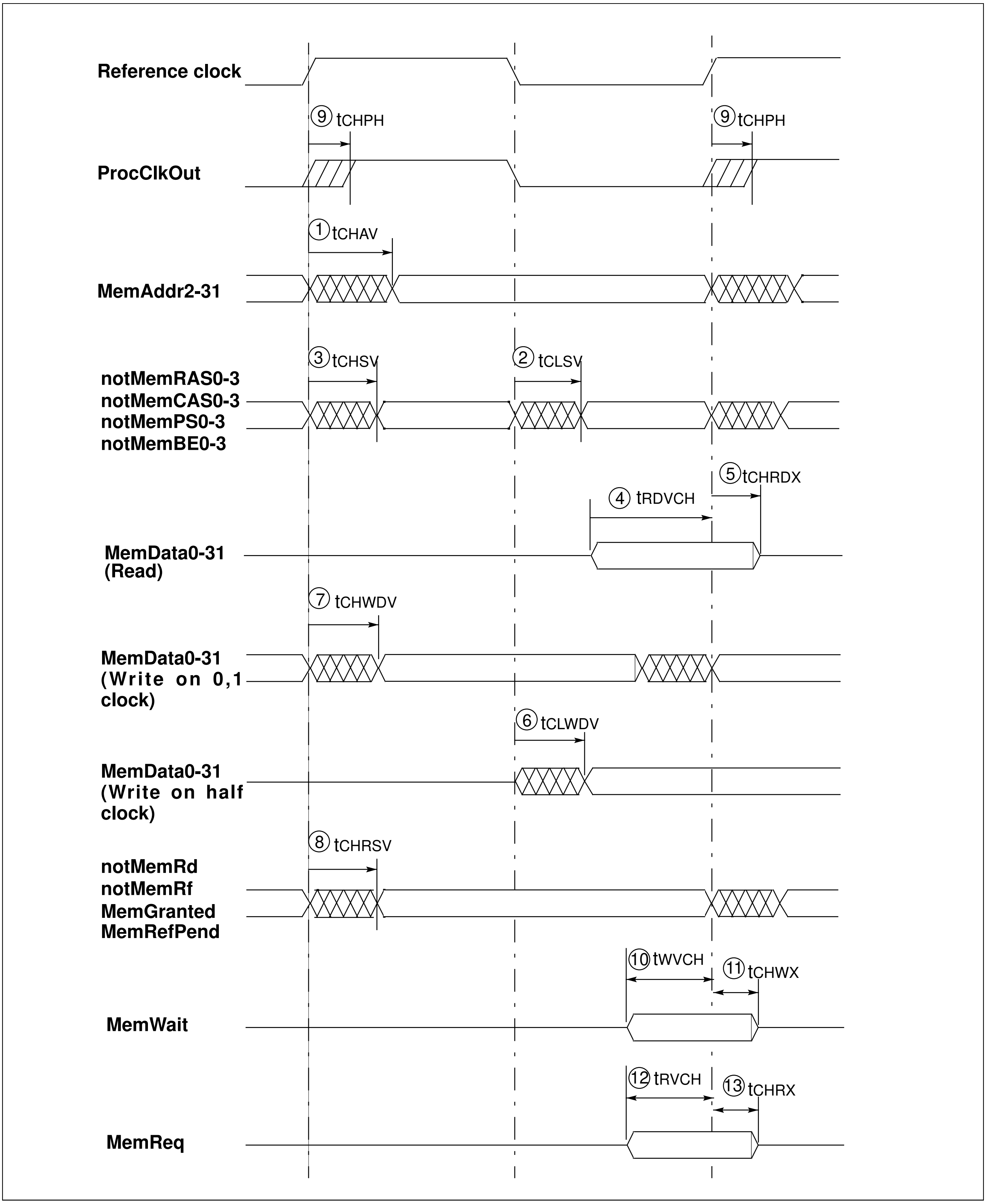


Figure 16.1 EMI timings

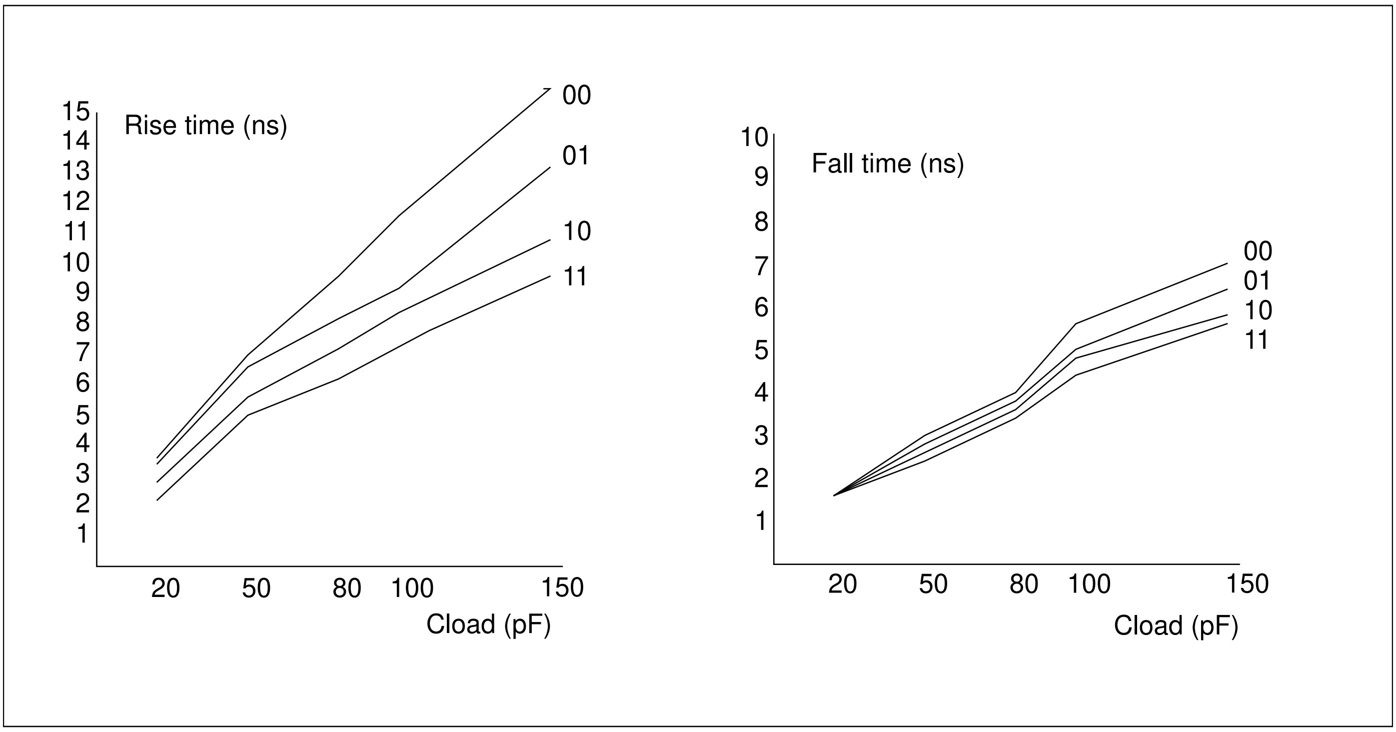


Figure 16.2 Rise and fall times for **MemData0-31** pins for different pad drive strengths

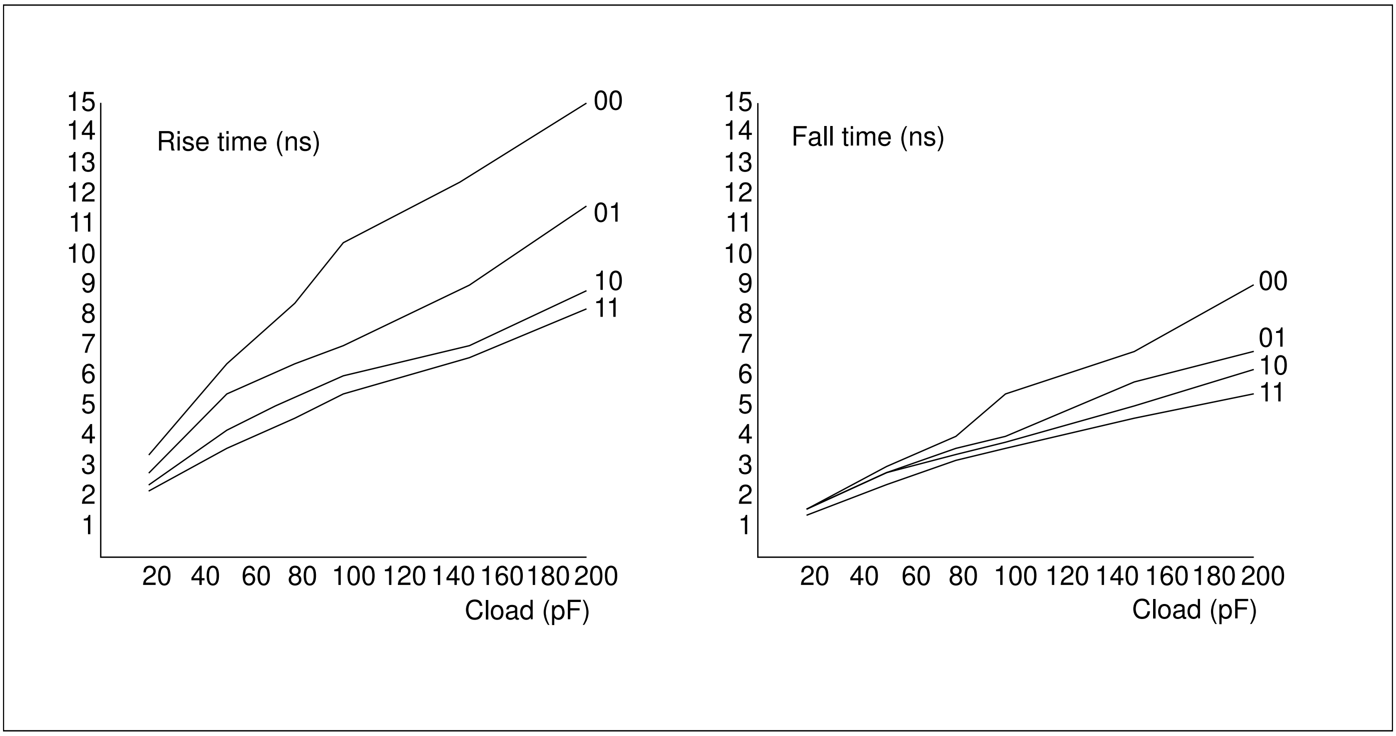


Figure 16.3 Rise and fall times for **MemAddr2-31** and strobe pins for different pad drive strengths

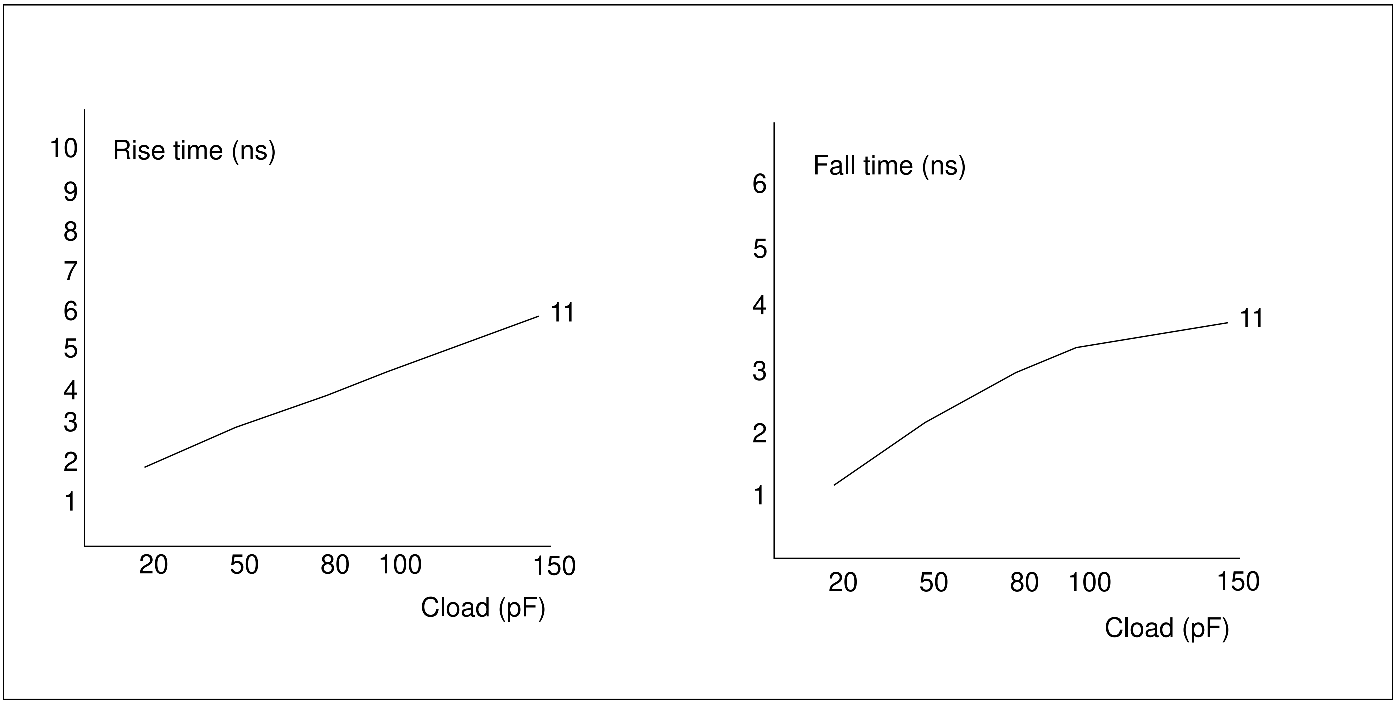


Figure 16.4 Rise and fall times for **ProcClkOut** at pad drive strength level 3

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

16.2 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 16.2 Link timings

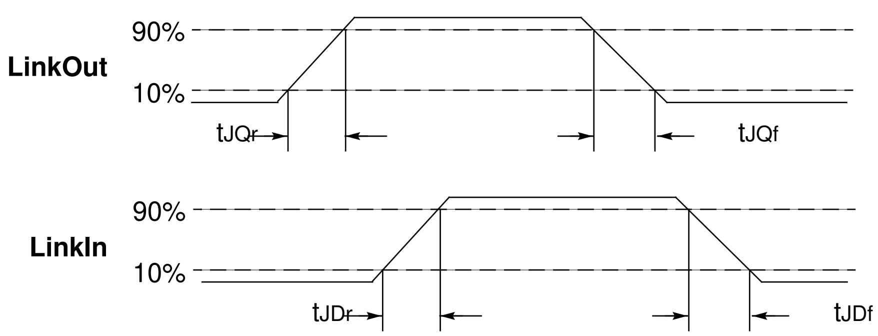


Figure 16.5 Link timings

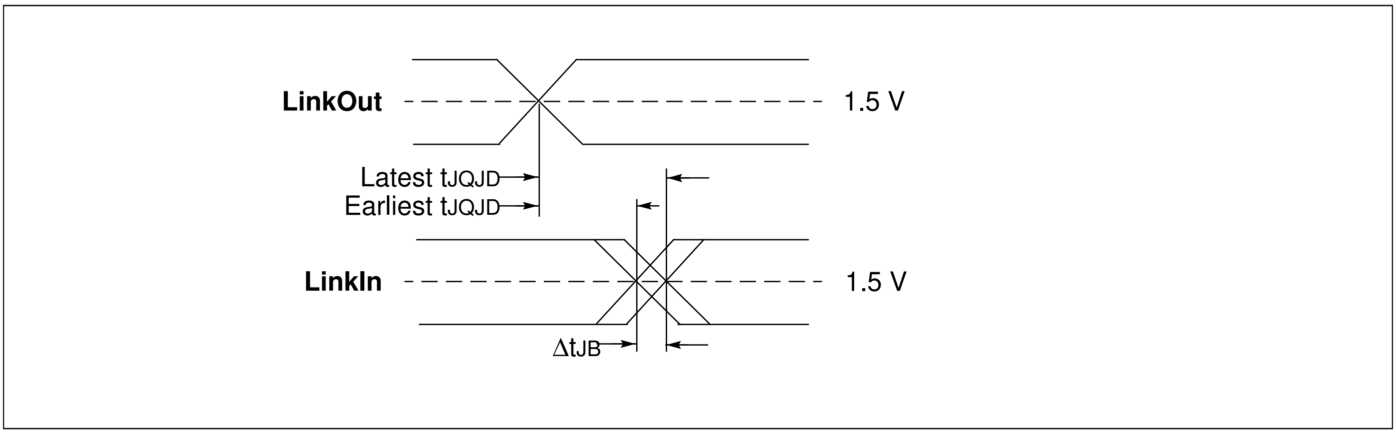


Figure 16.6 Buffered Link timings

16.3 Reset and Analyse timings

Symbol	Parameter	Min	Nom	Max	Units	Notes
t_{RHRL}	**notRST** pulse width low	8			ClockIn	
t_{RHRL}	**CPUReset** pulse width high	1			ClockIn	
t_{AHRH}	**CPUAnalyse** setup before **CPUReset**	3			ms	1
t_{RLAL}	**CPUAnalyse** hold after **CPUReset** end	1			ClockIn	

Notes

1 When **ResetRespOut** is high.

Table 16.3 Reset and Analyse timings

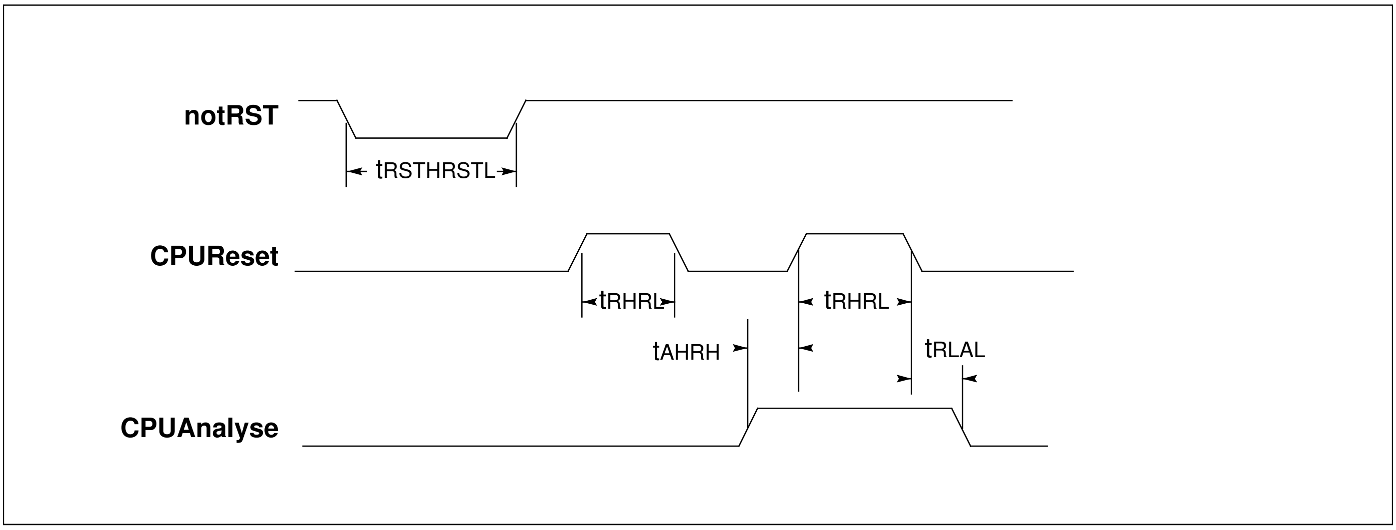


Figure 16.7 Reset and Analyse timings

SGS-THOMSON MICROELECTRONICS

16.4 Event timings

Symbol	Parameter	Min	Max	Units	Notes
tVHKH	**EventReq** response	0		ns	
tKHVL	**EventReq** hold	0		ns	
tVLKL	Delay before removal of **EventAck**	0		ns	
tKLVH	Delay before re-assertion of **EventReq**	0		ns	
tKHEWL	**EventAck** to end of **EventWaiting**	0		ns	

Table 16.4 Event timings

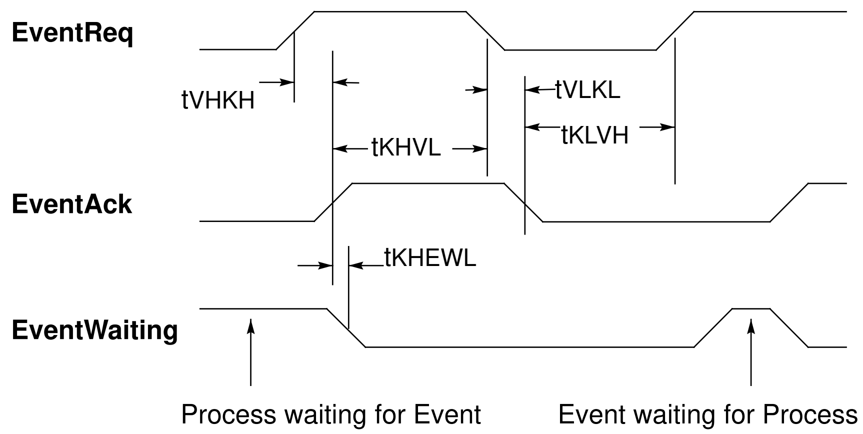


Figure 16.8 Event timings

16.5 Clock timings

16.5.1 ClockIn and LinkClockIn timings

Symbol	Parameter	Min	Nom	Max	Units	Notes
t_{DCLDCH}	**ClockIn** and **LinkClockIn** pulse width low for PLL operation	40			ns	
t_{DCHDCL}	**ClockIn** and **LinkClockIn** pulse width high for PLL operation	40			ns	
t_{DCLDCL}	**ClockIn** and **LinkClockIn** period for PLL operation		200		ns	1, 2
t_{DCr}	**ClockIn** and **LinkClockIn** rise time for PLL operation			10	ns	3
t_{DCf}	**ClockIn** and **LinkClockIn** fall time for PLL operation			8	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 16.5 **ClockIn** and **LinkClockIn** timings

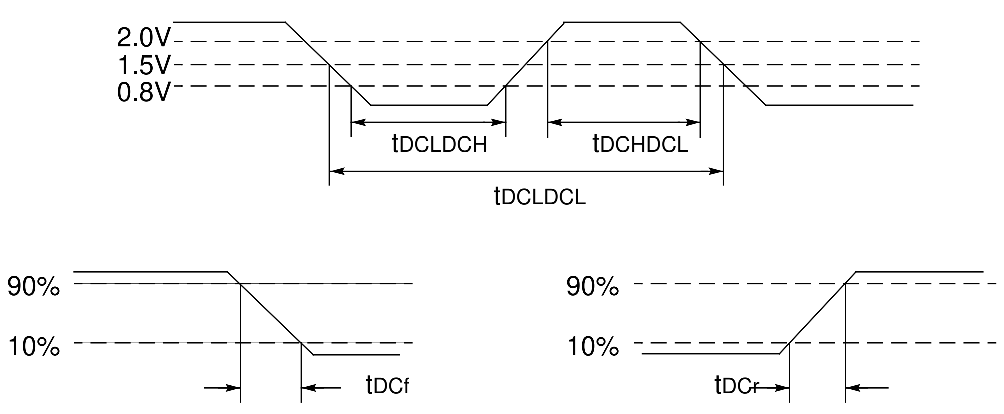


Figure 16.9 **ClockIn** and **LinkClockIn** timings

16.5.2 ProcClkOut timings

Symbol	Parameter	Min	Max	Units	Notes
tPCLPCL	**ProcClkOut** period	22.5	27.5	ns	
tPCHPCL	**ProcClkOut** pulse width high	10	15	ns	
tPCLPCH	**ProcClkOut** pulse width low	10	15	ns	
tPCstab	**ProcClkOut** stability		8	%	1

Notes

1 Stability is the variation of cycle periods between two consecutive cycles, measured at corresponding points on the cycles.

Table 16.6 **ProcClkOut** timings

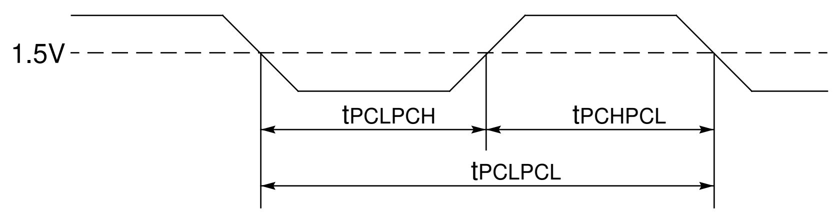


Figure 16.10 **ProcClkOut** timings

16.6 TAP timings

The TAP will function at 5 MHz **TCK**, with T_{setup} = 10ns and T_{hold} = 10ns for all inputs, and T_{prop} = 50ns for all outputs. All other electrical characteristics of the TAP pins are as defined in Chapter 15 on page 79.

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

Table 16.7 TAP timings

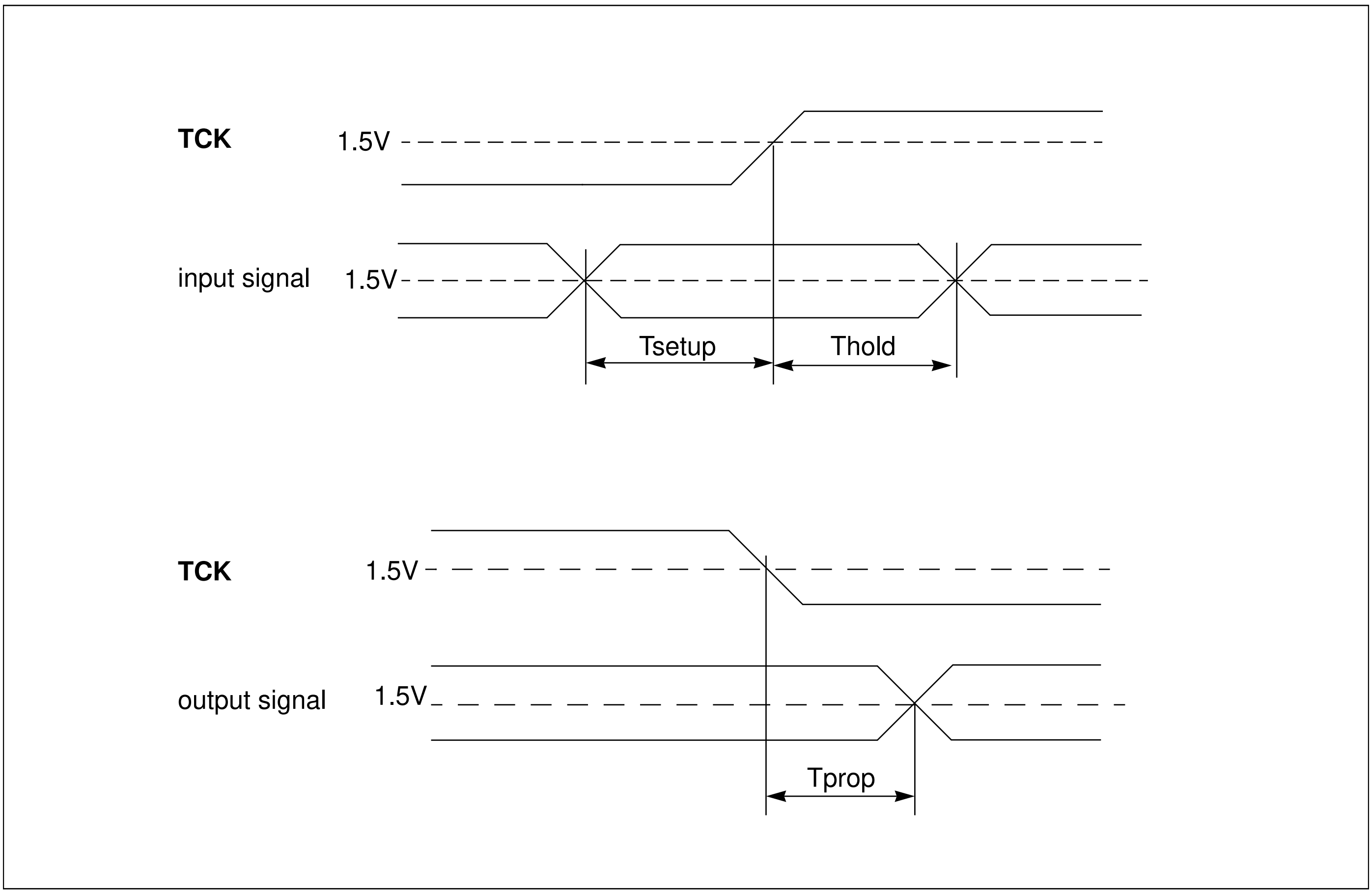


Figure 16.11 TAP timings

17 Pin designations

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

Supplies

Pin	In/Out	Function
VDD		Power supply
GND		Ground

Table 17.1 ST20450 supply pins

Clocks

Pin	In/Out	Function
ClockIn	in	System input clock - PLL or TimesOneMode
LinkClockIn	in	Link input clock - PLL or TimesOneMode
LPClockIn	in	Low power input clock
LPOsc	in/out	Low power clock oscillator
LPIn	in	Low power control
LPOut	out	Low power status
ProcClkOut	out	Processor clock

Table 17.2 ST20450 clocks and low power pins

System services

Pin	In/Out	Function
ProcSpeed0-2	in	Processor speed selectors
ResetRespOut	out	Reset response output
notRST	in	Reset
CPUReset	in	System reset
CPUAnalyse	in	Error analysis

Table 17.3 ST20450 system services pins

CPU

Pin	In/Out	Function
ErrorOut	out	Error indicator
ErrorIn	in	Error daisy-chain input
Debug0-7	out	Debug output
DebugIn	in	Debug input

Table 17.4 ST20450 CPU pins

Interrupts

Pin	In/Out	Function
Interrupt0-7	in	Interrupt

Table 17.5 ST20450 interrupt pins

Memory

Pin	In/Out	Function
MemAddr2-31	out	Address bus
MemData0-31	in/out	Data bus. **Data0** is the least significant bit (LSB) and **Data31** is the most significant bit (MSB).
notMemRd	out	Read strobe
MemReq	in	Direct memory access request
MemGranted	out	Direct memory access granted
MemRefPend	out	Dynamic memory refresh cycle is pending
notMemRf	out	Dynamic memory refresh indicator
MemWait	in	Memory cycle extender
notMemCAS0-3	out	CAS strobes - one per bank
notMemRAS0-3	out	RAS strobes - one per bank
notMemPS0-3	out	Programmable strobes - one per bank
notMemBE0-3	out	Byte enable strobes - one per bank
BootSrce0-1	in	Boot from ROM or from link
DisableRAM	in	Disables internal SRAM

Table 17.6 ST20450 memory pins

Links

Pin	In/Out	Function
LinkIn0-3	in	Four serial data input channels
LinkOut0-3	out	Four serial data output channels
Link0Special	in	Select special speed for Link 0
LinkSpeed0-1	in	Link speed selectors

Table 17.7 ST20450 link pins

Event

Pin	In/Out	Function
EventReq	in	Event request
EventAck	out	Event request acknowledge
EventWaiting	out	Event input requested by software

Table 17.8 ST20450 event pins

Test Access Port (TAP)

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

Table 17.9 ST20450 TAP pins

Miscellaneous

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

Table 17.10 ST20450 miscellaneous pins

18 Package specifications

The ST20450 is available in a 208 pin plastic quad flat pack (PQFP) package.

18.1 ST20450 package pinout

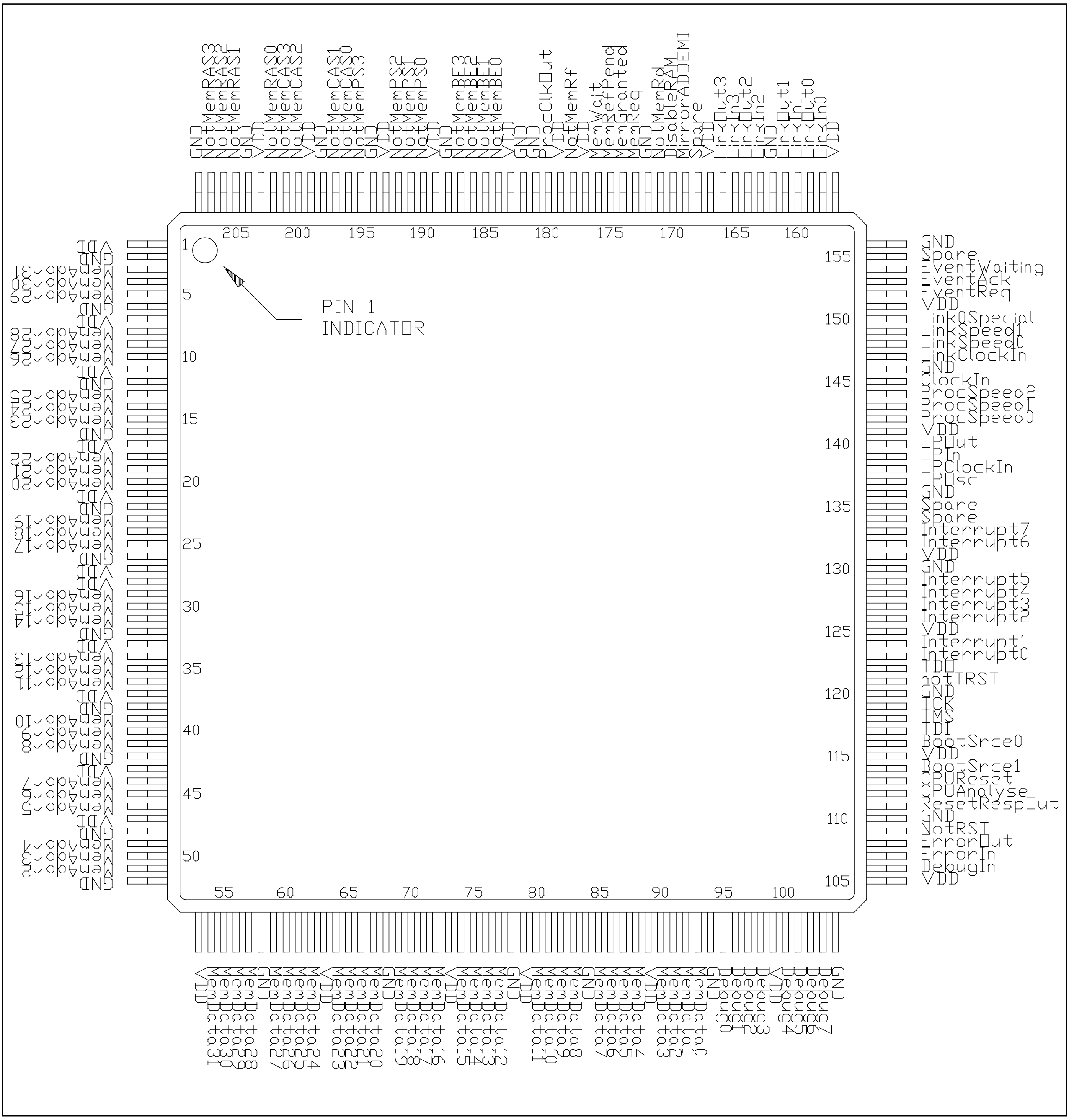


Figure 18.1 ST20450 208 pin PQFP package pinout

18.2 ST20450 208 pin PQFP package dimensions

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

Notes

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

Table 18.1 208 pin PQFP package dimensions

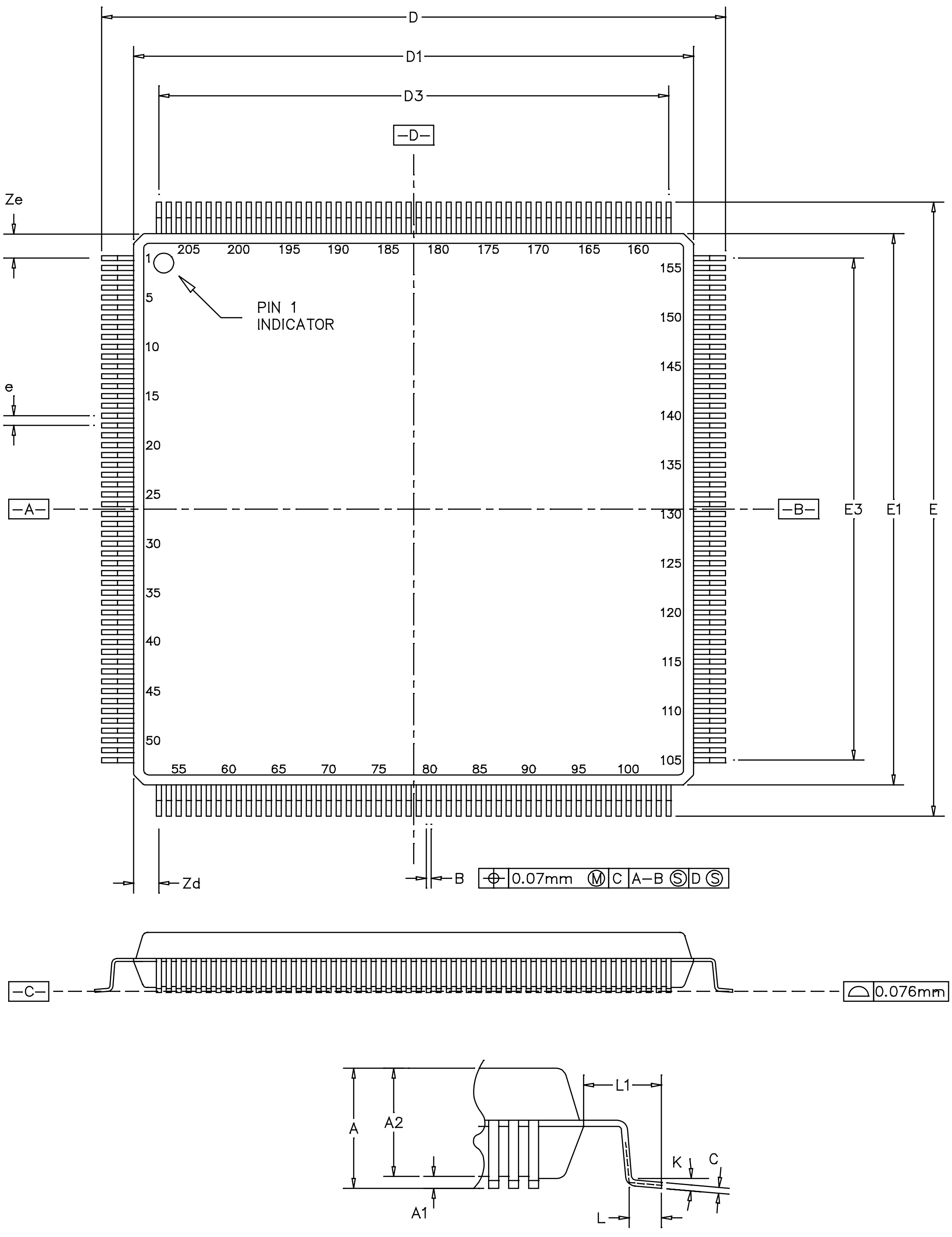


Figure 18.2 ST20450 208 pin PQFP package dimensions

18.3 ST20450 208 pin PQFP package thermal data

Maximum, still air thermal resistance is 55°C/W for the die in a 208 PQFP, with copper leadframe.

Junction to case thermal resistance, measured to the centre of the upper side of the case, is 25°C/W.

Given a maximum operating junction temperature of 100°C the following maximum power conditions apply;

Still air at 35°C	2.5 W
Still air at 85°C	0.83 W
Case held at 85°C	1.85 W

Under certain operating conditions, the ST20450 device can dissipate up to 1.85 W. External thermal management is recommended, to ensure optimum performance and reliability.

19 Ordering information

Device	Package
ST20450-X40S	208 pin plastic quad flatpack (PQFP)

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

Appendix A Boundary scan description language (BSDL) file

```
--Version 0.2
--P J Dickinson, 30-08-95

entity ST20450A is

 generic (PHYSICAL_PIN_MAP : string := "PQFP208");

 port (VDD: linkage bit_vector (0 to 29);
       GND: linkage bit_vector (0 to 31);

       spare: linkage bit_vector (0 to 3);

       ClockIn, LinkClockIn, LPClockIn: in bit;
       LPClockOsc: linkage bit;
        -- this inout is a clock, and therefore has no BS cell
       LPIn: in bit;
       LPOut, ProcClkOut: out bit;

       ProcSpeed: in bit_vector (0 to 2);
       ResetRespOut: out bit;
       notRST, CPUReset, CPUAnalyse: in bit;

       ErrorOut: out bit;
       ErrorIn: in bit;
       DebugOut: out bit_vector (0 to 7);
       DebugIn: in bit;

       Interrupt: in bit_vector (0 to 7);

       MemAddr: out bit_vector (2 to 31);
       MemData: inout bit_vector (0 to 31);
       notMemRd: out bit;
       MemReq: in bit;
       MemGranted, MemRefPend, notMemRf: out bit;
       MemWait: in bit;
       notMemCAS, notMemRAS, notMemPS, notMemBE: out bit_vector (0 to 3);
       BootSrce: in bit_vector (0 to 1);
       DisableRAM: in bit;

       LinkIn: in bit_vector (0 to 3);
       LinkOut: out bit_vector (0 to 3);
       Link0Special: in bit;
       LinkSpeed: in bit_vector (0 to 1);

       EventReq: in bit;
       EventAck: out bit;
       EventWaiting: linkage bit;
        -- This output has no BS cell due to a bug

       TDI: in bit;
       TDO: out bit;
       TMS, TCK, notTRST: in bit;
```

```
      MirrorADDEMI: in bit );

 use STD_1149_1_1990.all;

 attribute PIN_MAP of ST20450A : entity is PHYSICAL_PIN_MAP;

 constant PQFP208: PIN_MAP_STRING :=


"memaddr:(3,4,5,8,9,10,13,14,15,18,19,20,23,24,25,29,30,31,34,35,36,39,40,41,44,45,46,
49,50,51)," &

"memdata:(54,55,56,57,59,60,61,62,64,65,66,67,69,70,71,72,74,75,76,77,80,81,82,83,85,8
6,87,88,90,91,92,93)," &
  "debugout:(95,96,97,98,100,101,102,103)," &
  "debugin:106," &
  "errorin:107," &
  "errorout:108," &
  "notrst:109," &
  "resetrespout:111," &
  "cpuanalyse:112," &
  "cpureset:113," &
  "bootsrce:(114,116)," &
  "tdi:117," &
  "tms:118," &
  "tck:119," &
  "nottrst:121," &
  "tdo:122," &
  "interrupt:(123,124,126,127,128,129,132,133)," &
  "lpclockosc:137," &
  "lpclockin:138," &
  "lpin:139," &
  "lpout:140," &
  "procspeed:(142,143,144)," &
  "clockin:145," &
  "linkclockin:147," &
  "linkspeed:(148,149)," &
  "link0special:150," &
  "eventreq:152," &
  "eventack:153," &
  "eventwaiting:154," &
  "linkin:(158,160,163,165)," &
  "linkout:(159,161,164,166)," &
  "mirroraddemi:169," &
  "disableram:170," &
  "notmemrd:171," &
  "memreq:173," &
  "memgranted:174," &
  "memrefpend:175," &
  "memwait:176," &
  "notmemrf:178," &
  "procclkout:180," &
  "notmembe:(184,185,186,187)," &
  "notmemps:(190,191,192,195)," &
  "notmemcas:(196,197,200,201)," &
  "notmemras:(202,205,206,207)," &
```

```
"vdd:(11,17,21,27,28,33,37,43,47,53,63,73,79,89,99,105,115,125,131,141,151,157,167,177
,179,183,189,193,199,203)," &

"gnd:(2,6,12,16,22,26,32,38,42,48,52,58,68,78,84,94,104,110,120,130,136,146,156,162,17
2,181,182,188,194,198,204,208)," &
  "spare:(134,135,155,168)";

 attribute TAP_SCAN_IN    of TDI : signal is true;
 attribute TAP_SCAN_MODE  of TMS : signal is true;
 attribute TAP_SCAN_OUT   of TDO : signal is true;
 attribute TAP_SCAN_CLOCK of TCK : signal is (10.0e6, BOTH);
 attribute TAP_SCAN_RESET of notTRST: signal is true;

 attribute INSTRUCTION_LENGTH of ST20450A : entity is 5;

 attribute INSTRUCTION_OPCODE of ST20450A : entity is
  "BYPASS          (11111), " &
  "EXTEST          (00000), " &
  "IDCODE          (00001), " &
  "SAMPLE          (00010)  " ;

 attribute INSTRUCTION_CAPTURE of ST20450A : entity is "00001";

 attribute IDCODE_REGISTER of ST20450A : entity is "00000101000000000000000000010001";

 attribute BOUNDARY_LENGTH of ST20450A : entity is 181;

 attribute BOUNDARY_REGISTER of ST20450A : entity is
 -- num cell  port                function safe [ccell disval rslt]
  "   0 (BC_1,  notmemras(3)       ,   output2, X)," &
  "   1 (BC_1,  notmemras(2)       ,   output2, X)," &
  "   2 (BC_1,  notmemras(1)       ,   output2, X)," &
  "   3 (BC_1,  notmemras(0)       ,   output2, X)," &
  "   4 (BC_1,  notmemcas(3)       ,   output2, X)," &
  "   5 (BC_1,  notmemcas(2)       ,   output2, X)," &
  "   6 (BC_1,  notmemcas(1)       ,   output2, X)," &
  "   7 (BC_1,  notmemcas(0)       ,   output2, X)," &
  "   8 (BC_1,  notmemps(3)        ,   output2, X)," &
  "   9 (BC_1,  notmemps(2)        ,   output2, X)," &
  "  10 (BC_1,  notmemps(1)        ,   output2, X)," &
  "  11 (BC_1,  notmemps(0)        ,   output2, X)," &
  "  12 (BC_1,  notmembe(3)        ,   output2, X)," &
  "  13 (BC_1,  notmembe(2)        ,   output2, X)," &
  "  14 (BC_1,  notmembe(1)        ,   output2, X)," &
  "  15 (BC_1,  notmembe(0)        ,   output2, X)," &
  "  16 (BC_1,  procclkout         ,   output2, X)," &
  "  17 (BC_1,  *                  ,  internal, X)," &
  "  18 (BC_1,  notmemrf           ,   output2, X)," &
  "  19 (BC_1,  memwait            ,     input, X)," &
  "  20 (BC_1,  *                  ,  internal, X)," &
  "  21 (BC_1,  *                  ,  internal, X)," &
  "  22 (BC_1,  memrefpend         ,   output2, X)," &
  "  23 (BC_1,  *                  ,  internal, X)," &
  "  24 (BC_1,  memgranted         ,   output2, X)," &
  "  25 (BC_1,  memreq             ,     input, X)," &
  "  26 (BC_1,  *                  ,  internal, X)," &
  "  27 (BC_1,  *                  ,  internal, X)," &
  "  28 (BC_1,  notmemrd           ,   output2, X)," &
```

```
"  29 (BC_1,  disableram         ,     input, X)," &
"  30 (BC_1,  *                  ,  internal, X)," &
"  31 (BC_1,  mirroraddemi       ,     input, X)," &
"  32 (BC_1,  *                  ,  internal, X)," &
"  33 (BC_1,  linkout(3)         ,   output2, X)," &
"  34 (BC_1,  linkin (3)         ,     input, X)," &
"  35 (BC_1,  linkout(2)         ,   output2, X)," &
"  36 (BC_1,  linkin (2)         ,     input, X)," &
"  37 (BC_1,  linkout(1)         ,   output2, X)," &
"  38 (BC_1,  linkin (1)         ,     input, X)," &
"  39 (BC_1,  linkout(0)         ,   output2, X)," &
"  40 (BC_1,  linkin (0)         ,     input, X)," &
 -- " (BC_1,  eventwaiting       ,   output2, X)," &
 -- no BS cell here due to bug
"  41 (BC_1,  eventack           ,   output2, X)," &
"  42 (BC_1,  eventreq           ,     input, X)," &
"  43 (BC_1,  link0special       ,     input, X)," &
"  44 (BC_1,  linkspeed(1)       ,     input, X)," &
"  45 (BC_1,  linkspeed(0)       ,     input, X)," &
"  46 (BC_1,  linkclockin        ,     input, X)," &
"  47 (BC_1,  clockin            ,     input, X)," &
"  48 (BC_1,  procspeed(2)       ,     input, X)," &
"  49 (BC_1,  procspeed(1)       ,     input, X)," &
"  50 (BC_1,  procspeed(0)       ,     input, X)," &
"  51 (BC_1,  lpout              ,   output2, X)," &
"  52 (BC_1,  lpin               ,     input, X)," &
"  53 (BC_1,  lpclockin          ,     input, X)," &
"  54 (BC_1,  interrupt(7)       ,     input, X)," &
"  55 (BC_1,  interrupt(6)       ,     input, X)," &
"  56 (BC_1,  interrupt(5)       ,     input, X)," &
"  57 (BC_1,  interrupt(4)       ,     input, X)," &
"  58 (BC_1,  interrupt(3)       ,     input, X)," &
"  59 (BC_1,  interrupt(2)       ,     input, X)," &
"  60 (BC_1,  interrupt(1)       ,     input, X)," &
"  61 (BC_1,  interrupt(0)       ,     input, X)," &
"  62 (BC_1,  bootsrce(0)        ,     input, X)," &
"  63 (BC_1,  bootsrce(1)        ,     input, X)," &
"  64 (BC_1,  cpureset           ,     input, X)," &
"  65 (BC_1,  cpuanalyse         ,     input, X)," &
"  66 (BC_1,  resetrespout       ,   output2, X)," &
"  67 (BC_1,  notrst             ,     input, X)," &
"  68 (BC_1,  errorout           ,   output2, X)," &
"  69 (BC_1,  errorin            ,     input, X)," &
"  70 (BC_1,  debugin            ,     input, X)," &
"  71 (BC_1,  debugout(7)        ,   output2, X)," &
"  72 (BC_1,  debugout(6)        ,   output2, X)," &
"  73 (BC_1,  debugout(5)        ,   output2, X)," &
"  74 (BC_1,  debugout(4)        ,   output2, X)," &
"  75 (BC_1,  debugout(3)        ,   output2, X)," &
"  76 (BC_1,  debugout(2)        ,   output2, X)," &
"  77 (BC_1,  debugout(1)        ,   output2, X)," &
"  78 (BC_1,  debugout(0)        ,   output2, X)," &
"  79 (BC_1,  memdata( 0)        ,     input, X)," &
"  80 (BC_1,  memdata( 0)        ,   output3, X, 87, 0, Z)," &
"  81 (BC_1,  memdata( 1)        ,     input, X)," &
"  82 (BC_1,  memdata( 1)        ,   output3, X, 87, 0, Z)," &
"  83 (BC_1,  memdata( 2)        ,     input, X)," &
"  84 (BC_1,  memdata( 2)        ,   output3, X, 87, 0, Z)," &
```

```
"  85 (BC_1,  memdata( 3)        ,     input, X)," &
"  86 (BC_1,  memdata( 3)        ,   output3, X, 87, 0, Z)," &
"  87 (BC_1,  *                  ,   control, 0)," & -- mdata 0-7
"  88 (BC_1,  memdata( 4)        ,     input, X)," &
"  89 (BC_1,  memdata( 4)        ,   output3, X, 87, 0, Z)," &
"  90 (BC_1,  memdata( 5)        ,     input, X)," &
"  91 (BC_1,  memdata( 5)        ,   output3, X, 87, 0, Z)," &
"  92 (BC_1,  memdata( 6)        ,     input, X)," &
"  93 (BC_1,  memdata( 6)        ,   output3, X, 87, 0, Z)," &
"  94 (BC_1,  memdata( 7)        ,     input, X)," &
"  95 (BC_1,  memdata( 7)        ,   output3, X, 87, 0, Z)," &
"  96 (BC_1,  memdata( 8)        ,     input, X)," &
"  97 (BC_1,  memdata( 8)        ,   output3, X,  104, 0, Z)," &
"  98 (BC_1,  memdata( 9)        ,     input, X)," &
"  99 (BC_1,  memdata( 9)        ,   output3, X,  104, 0, Z)," &
" 100 (BC_1,  memdata(10)        ,     input, X)," &
" 101 (BC_1,  memdata(10)        ,   output3, X,  104, 0, Z)," &
" 102 (BC_1,  memdata(11)        ,     input, X)," &
" 103 (BC_1,  memdata(11)        ,   output3, X,  104, 0, Z)," &
" 104 (BC_1,  *                  ,   control, 0)," & -- mdata 8-15
" 105 (BC_1,  memdata(12)        ,     input, X)," &
" 106 (BC_1,  memdata(12)        ,   output3, X,  104, 0, Z)," &
" 107 (BC_1,  memdata(13)        ,     input, X)," &
" 108 (BC_1,  memdata(13)        ,   output3, X,  104, 0, Z)," &
" 109 (BC_1,  memdata(14)        ,     input, X)," &
" 110 (BC_1,  memdata(14)        ,   output3, X,  104, 0, Z)," &
" 111 (BC_1,  memdata(15)        ,     input, X)," &
" 112 (BC_1,  memdata(15)        ,   output3, X,  104, 0, Z)," &
" 113 (BC_1,  *                  ,   control, 0)," & -- mdata 16-23
" 114 (BC_1,  memdata(16)        ,     input, X)," &
" 115 (BC_1,  memdata(16)        ,   output3, X,  113, 0, Z)," &
" 116 (BC_1,  memdata(17)        ,     input, X)," &
" 117 (BC_1,  memdata(17)        ,   output3, X,  113, 0, Z)," &
" 118 (BC_1,  memdata(18)        ,     input, X)," &
" 119 (BC_1,  memdata(18)        ,   output3, X,  113, 0, Z)," &
" 120 (BC_1,  memdata(19)        ,     input, X)," &
" 121 (BC_1,  memdata(19)        ,   output3, X,  113, 0, Z)," &
" 122 (BC_1,  memdata(20)        ,     input, X)," &
" 123 (BC_1,  memdata(20)        ,   output3, X,  113, 0, Z)," &
" 124 (BC_1,  memdata(21)        ,     input, X)," &
" 125 (BC_1,  memdata(21)        ,   output3, X,  113, 0, Z)," &
" 126 (BC_1,  memdata(22)        ,     input, X)," &
" 127 (BC_1,  memdata(22)        ,   output3, X,  113, 0, Z)," &
" 128 (BC_1,  memdata(23)        ,     input, X)," &
" 129 (BC_1,  memdata(23)        ,   output3, X,  113, 0, Z)," &
" 130 (BC_1,  *                  ,   control, 0)," & -- mdata 24-31
" 131 (BC_1,  memdata(24)        ,     input, X)," &
" 132 (BC_1,  memdata(24)        ,   output3, X,  130, 0, Z)," &
" 133 (BC_1,  memdata(25)        ,     input, X)," &
" 134 (BC_1,  memdata(25)        ,   output3, X,  130, 0, Z)," &
" 135 (BC_1,  memdata(26)        ,     input, X)," &
" 136 (BC_1,  memdata(26)        ,   output3, X,  130, 0, Z)," &
" 137 (BC_1,  memdata(27)        ,     input, X)," &
" 138 (BC_1,  memdata(27)        ,   output3, X,  130, 0, Z)," &
" 139 (BC_1,  memdata(28)        ,     input, X)," &
" 140 (BC_1,  memdata(28)        ,   output3, X,  130, 0, Z)," &
" 141 (BC_1,  memdata(29)        ,     input, X)," &
" 142 (BC_1,  memdata(29)        ,   output3, X,  130, 0, Z)," &
```

```
  " 143 (BC_1,  memdata(30)         ,     input, X)," &
  " 144 (BC_1,  memdata(30)         ,   output3, X,  130, 0, Z)," &
  " 145 (BC_1,  memdata(31)         ,     input, X)," &
  " 146 (BC_1,  memdata(31)         ,   output3, X,  130, 0, Z)," &
  " 147 (BC_1,  memaddr( 2)         ,   output3, X,  150, 0, Z)," &
  " 148 (BC_1,  memaddr( 3)         ,   output3, X,  150, 0, Z)," &
  " 149 (BC_1,  memaddr( 4)         ,   output3, X,  150, 0, Z)," &
  " 150 (BC_1,  *                   ,   control, 0)," & -- maddr 2-7
  " 151 (BC_1,  memaddr( 5)         ,   output3, X,  150, 0, Z)," &
  " 152 (BC_1,  memaddr( 6)         ,   output3, X,  150, 0, Z)," &
  " 153 (BC_1,  memaddr( 7)         ,   output3, X,  150, 0, Z)," &
  " 154 (BC_1,  memaddr( 8)         ,   output3, X,  160, 0, Z)," &
  " 155 (BC_1,  memaddr( 9)         ,   output3, X,  160, 0, Z)," &
  " 156 (BC_1,  memaddr(10)         ,   output3, X,  160, 0, Z)," &
  " 157 (BC_1,  memaddr(11)         ,   output3, X,  160, 0, Z)," &
  " 158 (BC_1,  memaddr(12)         ,   output3, X,  160, 0, Z)," &
  " 159 (BC_1,  memaddr(13)         ,   output3, X,  160, 0, Z)," &
  " 160 (BC_1,  *                   ,   control, 0)," & -- maddr 8-15
  " 161 (BC_1,  memaddr(14)         ,   output3, X,  160, 0, Z)," &
  " 162 (BC_1,  memaddr(15)         ,   output3, X,  160, 0, Z)," &
  " 163 (BC_1,  memaddr(16)         ,   output3, X,  164, 0, Z)," &
  " 164 (BC_1,  *                   ,   control, 0)," & -- maddr 16-23
  " 165 (BC_1,  memaddr(17)         ,   output3, X,  164, 0, Z)," &
  " 166 (BC_1,  memaddr(18)         ,   output3, X,  164, 0, Z)," &
  " 167 (BC_1,  memaddr(19)         ,   output3, X,  164, 0, Z)," &
  " 168 (BC_1,  memaddr(20)         ,   output3, X,  164, 0, Z)," &
  " 169 (BC_1,  memaddr(21)         ,   output3, X,  164, 0, Z)," &
  " 170 (BC_1,  memaddr(22)         ,   output3, X,  164, 0, Z)," &
  " 171 (BC_1,  memaddr(23)         ,   output3, X,  164, 0, Z)," &
  " 172 (BC_1,  memaddr(24)         ,   output3, X,  174, 0, Z)," &
  " 173 (BC_1,  memaddr(25)         ,   output3, X,  174, 0, Z)," &
  " 174 (BC_1,  *                   ,   control, 0)," & -- maddr 24-31
  " 175 (BC_1,  memaddr(26)         ,   output3, X,  174, 0, Z)," &
  " 176 (BC_1,  memaddr(27)         ,   output3, X,  174, 0, Z)," &
  " 177 (BC_1,  memaddr(28)         ,   output3, X,  174, 0, Z)," &
  " 178 (BC_1,  memaddr(29)         ,   output3, X,  174, 0, Z)," &
  " 179 (BC_1,  memaddr(30)         ,   output3, X,  174, 0, Z)," &
  " 180 (BC_1,  memaddr(31)         ,   output3, X,  174, 0, Z)";

end ST20450A;
```