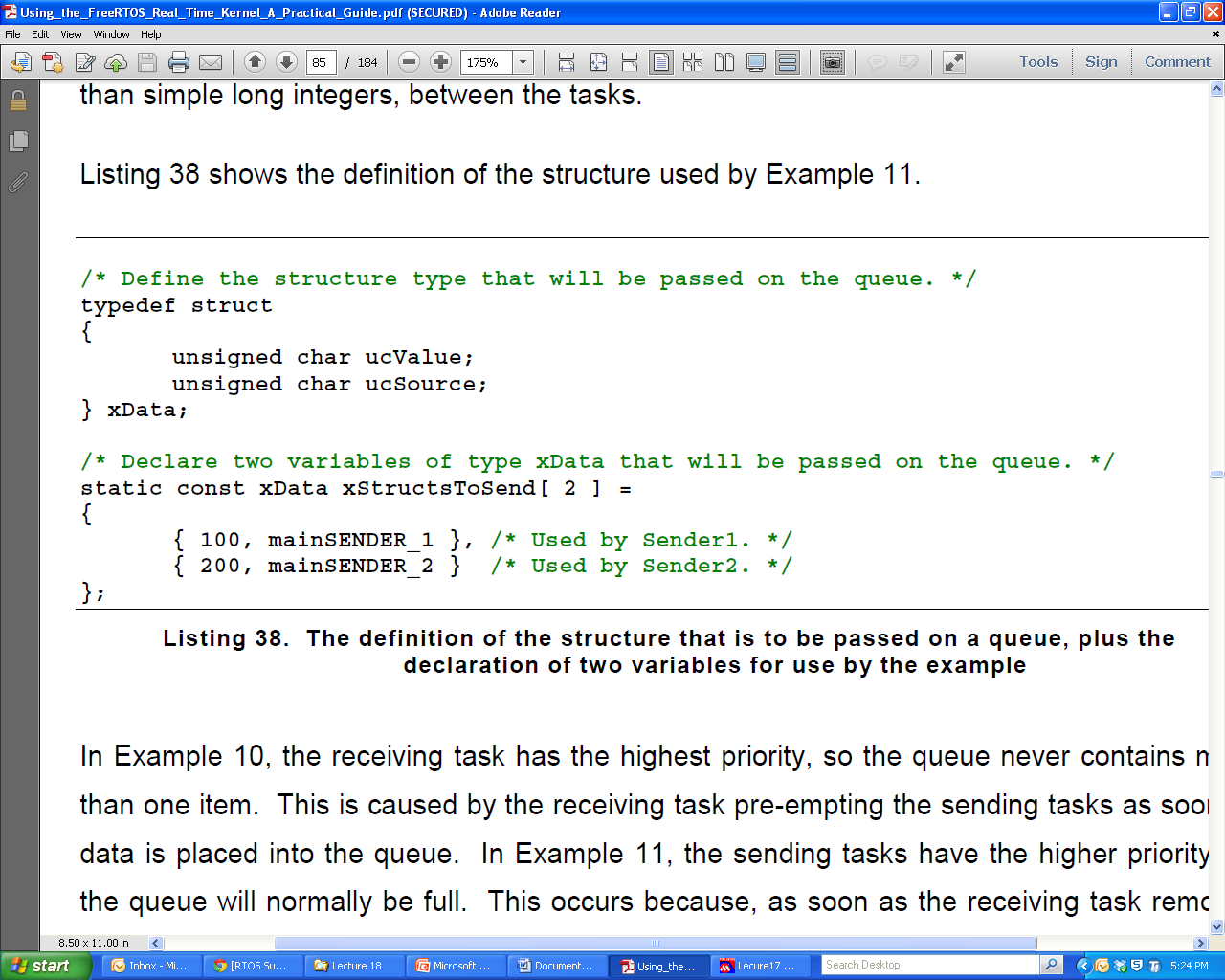
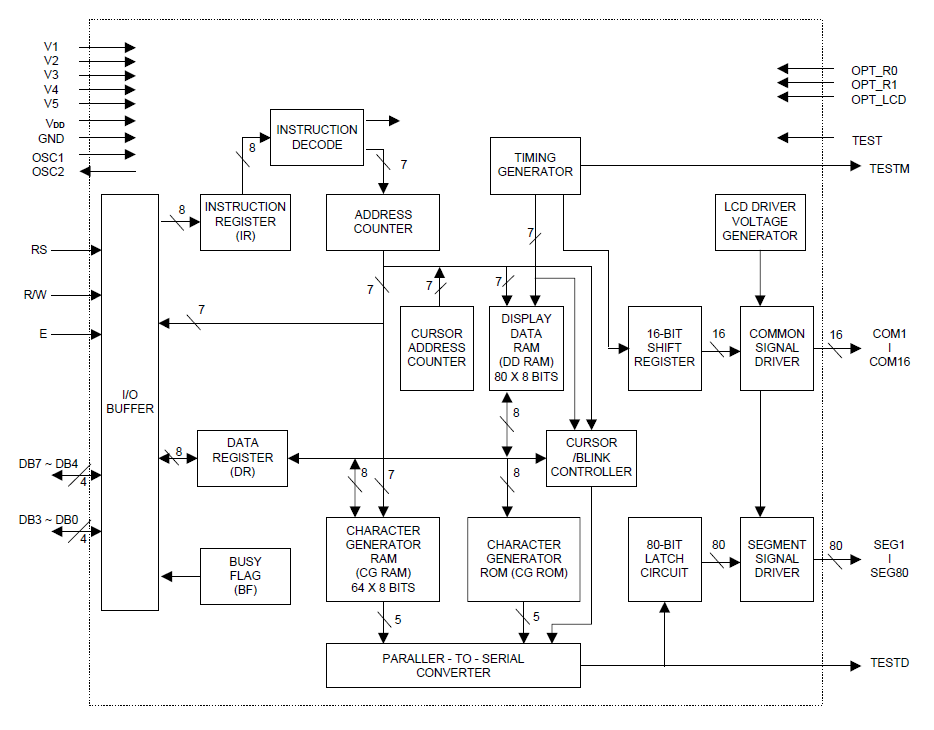
* **Task** – Is a single program process, strand, or section. It has a clear and distinct purpose and outcome. It is independent of other tasks, and it can’t call other tasks.
* **Multi-tasking** – Many tasks performed ideally simultaneously.
* **Task Priorities** – Task have priority, that is relative to the other tasks. Higher priority tasks run before lower priority tasks.
* **Deadline** – when the task must be done, this can be based upon times or events.
* ***A real Time System: A system operating in real time must be able to provide the correct results at the required time deadlines.***
* **Problems with Super loop approach:**
* *Loop execution time is not constant*.
* *Tasks interfere with each other.*
* *High priority tasks don’t get the attention they need*.
* **Operating System (OS):** a layer of SW that provides low level services and management of tasks or applications. This management consists of: scheduling the tasks, managing the context of the tasks, provide communication and synchronization between tasks and sharing and allocating system resources (e.g. peripherals) between the tasks.
* **RTOS:** an OS that allows one to specify constraints on the rate of process, that guarantees that these rate constraints will be met.
* **Why RTOS:**
* Abstraction away from timing, Maintainability / Extensibility, Modularity, Improved Efficiency
* Idle time, Flexible interrupt handling, Mixed processing requirement, Easier control and sharing of peripherals amongst tasks.
* **The interrupt and time tick ‘operating system’ works when**:
  + There are not too many tasks;
  + Task priorities can be accommodated in the structure;
  + Tasks are moderately well behaved, for example their requirement for CPU time is always reasonable, and interrupt-driven tasks don’t occur too often.
* If these limits are too much time to consider a Real Time Operating System (RTOS)
* **Scheduler**: Part of the kernel that decides which task should execute, when, and for how long. The scheduler can start and stop a task numerous times before the task finishes.
* **Scheduler Policy:** The algorithm in the scheduler that is used to “fairly” divide processor time between tasks.
* **Non-preemptive Scheduler** – Scheduler will start the highest priority task and then not switch tasks until current task completes or blocks itself. Sometimes called Co-operative scheduling
* **Preemptive Scheduler** – Will only let a task execute for a predefined time quantum or tick and then re-evaluate for the highest priority ready task. It will preempt the current task if a higher priority task is ready.
* **Fixed Priority** – The kernel can not change the priority of a task
* **Dynamic Priority** – The kernel can change the priority on the fly. For example aging of waiting tasks could cause them to increase in priority.
* ***Round Robin****:* If a scheduler is preemptive and if a number of tasks of the same priority are ready then each time tick or time Quantum a new ready task will be run
* ***Priority Scheduling****:* If a scheduler is non-preemptive then first come first serve each tasks runs to finish
* One way is to assign the highest priority to the most frequent (i.e. smallest period) task, called ***rate monotonic*** *scheduling*
* Another way is to assign the higher priority to the task that has the tightest (least time) deadline, called ***deadline monotonic scheduling***
* Consider: both rate and deadline in making priorities
* **Context of a Task:** All the information needed to be restored so that a task can be restarted this includes:
  + Program Counter
  + Status Register
  + Working Registers
  + Stack Pointers values
* **Context Switching:** The action of saving the context of one task and restoring the context of another task.
* Latency: The time it takes to do context switching
* At any instance only one Task can be running
* All other tasks are **either ready to run, blocked, or suspended**

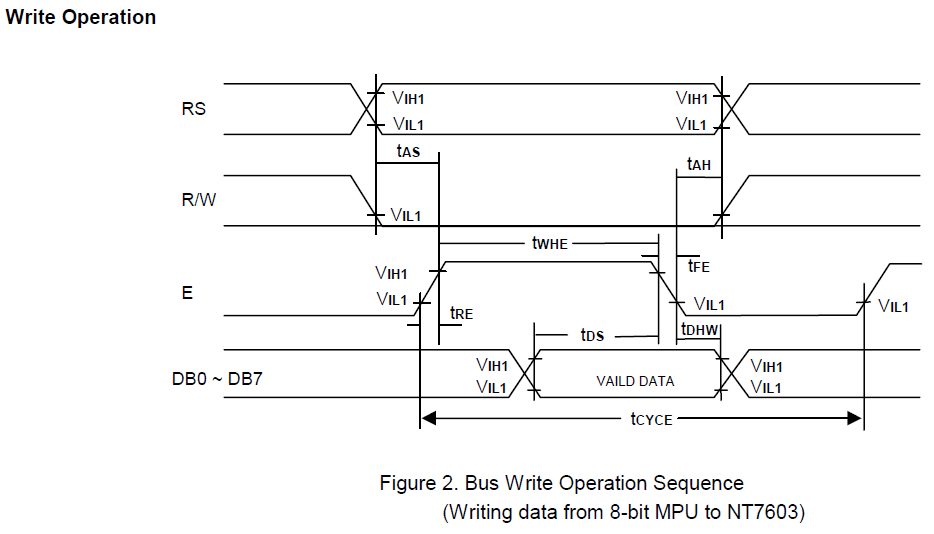


* Entered waiting for a **synchronization** event
  + External event like sensor activation
  + Internal event like completion of another tasks analysis
  + Waiting for a peripheral that is being used by another task.
* Entered waiting for a **temporal** event
  + A delay period expiring
  + An absolute time being reached
* ***Period* of a process:** a repeating time interval during which that process has to execute at least once.
* ***Execution deadline*:** the time before which a process must run to completion.
* **The Idle Task** will call a user supplied function to accomplish application specific functions
* To turn this feature on you must set the defined constant configUSE\_IDLE\_HOOK to be a value of 1
* The **idle task hook function** will be called automatically by the idle task once per iteration of the idle task loop.
* **FREERTOS**
* This is free open source code that can be found at freertos.org
* It is made for small to midsize processors of:
* 32 to 512 kbytes of ROM
* 16 to 256 kbytes or RAM
* FreeRTOS is a real time kernel or scheduler on top of which embedded applications can be built to meet hard real time requirements
* FreeRTOS can be commercially used in products under a modified **GNU General Purpose License (GPL)** with the provision that any modification to FreeRTOS is open source code, and you state to every customer that you are using FreeRTOS. The code is royalty free and has no warranty.
* The rest of the code as long as it uses just the provided API is still closed source.
* Other versions are for sale if you don’t want any open source, state that your using it, or want support.
* **Queues**
* A holds a finite number of fixed sized data items
* The **Length** of a queue is the max number of items it can hold
* Normally queues are First in First Out (FIFO) buffers
* A queue is its own object not owned by the creator of the task
* Multiple writers and readers of the queue are possible
* A queue having multiple writers is common, multiple readers is rare.
* If a task wants to read from the queue but the data has not been written yet, the task can block itself until the queue has been written to.
* If a task wants to write to the queue but the queue’s length has been reached then a task can block itself waiting for a queue read by another task to occur.
* In both cases a task can specify a max amount of time to wait.
* Prototype: void **taskYIELD**( void);
* Yield to another task of equal priority.
* This allows a task to yield with out waiting for the next time tick to preemptively change tasks
* If no other tasks at equal priority then the calling task gets the processor again.
* 

## LCD Display



* A Liquid Crystal Display operates on the principle that liquid crystal can be organized when a voltage is applied to it.
* A liquid crystal is sandwiched between two pieces of polarized glass. With no liquid crystal light can not pass through the polarized glass because of the polarization but with the crystal we get light to shine through (it looks gray)
* Light can then be reflected off a mirror an returned to the eye in a controlled manner. When voltage is applied to a LCD pixel it organize in such a way to make the pixel opaque to the light and it turn in to black
* The LCD we have is a pattern of rectangles. Each character space is made of 5 columns of 8 rows of these rectangles.
* **CGR** allows you to define your own bit pattern for a character cell. Your can store up to 8 characters of custom graphics.
* There are **64 addresses (0x40-0x7F) each address describes one row of the characte**r. Address 0x40 is the top row of the first character and 0x47 will be the bottom most row of the first character. Address 0x48 will start the top row of character 2 and so on.
* Writing the code 0-8 in DDR will display one of the 8 CGR defined characters.



tRE and tFE are the rise and fall times maximums they can be no longer than 25 ns.

tas is the RS and R/W setup time and a minimum of 60 ns is needed.

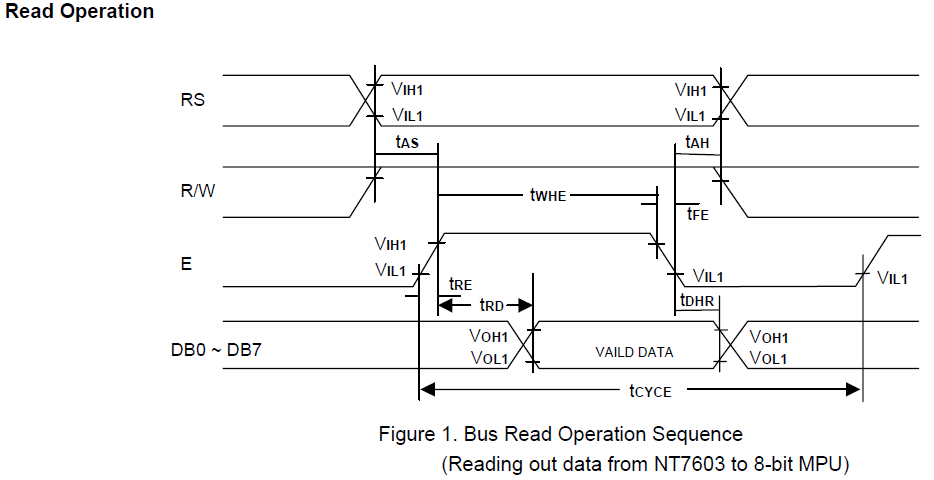
twhe is the minimum time that enable must be true it is 300 ns.

tds is the data setup time at a minimum it must be 150 ns

Tdhw is the data hold time and it must be at least 10 ns.

Tah is the hold time for RS and R/W it must at least 10ns

Tcycle is the minimum cycle time for a write that is 500 ns



RS = high for data register read RS= low for busy flag read

tRE and tFE are the rise and fall times maximums they can be no longer than 25 ns.

tasa is the RS and R/W setup time and a minimum of 60 ns is needed.

twhe is the minimum time that enable must be true it is 300 ns.

trd is read data output delay and it a maximum of 190ns

Tdhr is the data hold time and it will be at a minimum of 20 ns.

Tah is the hold time for RS and R/W it must at least 10ns

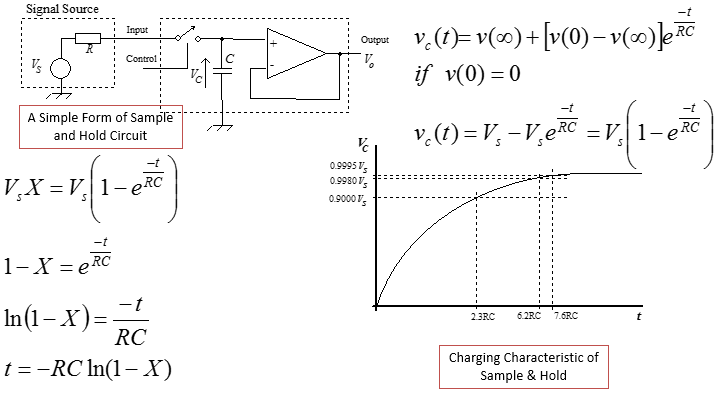
Tcycle is a minimum of 500 ns.

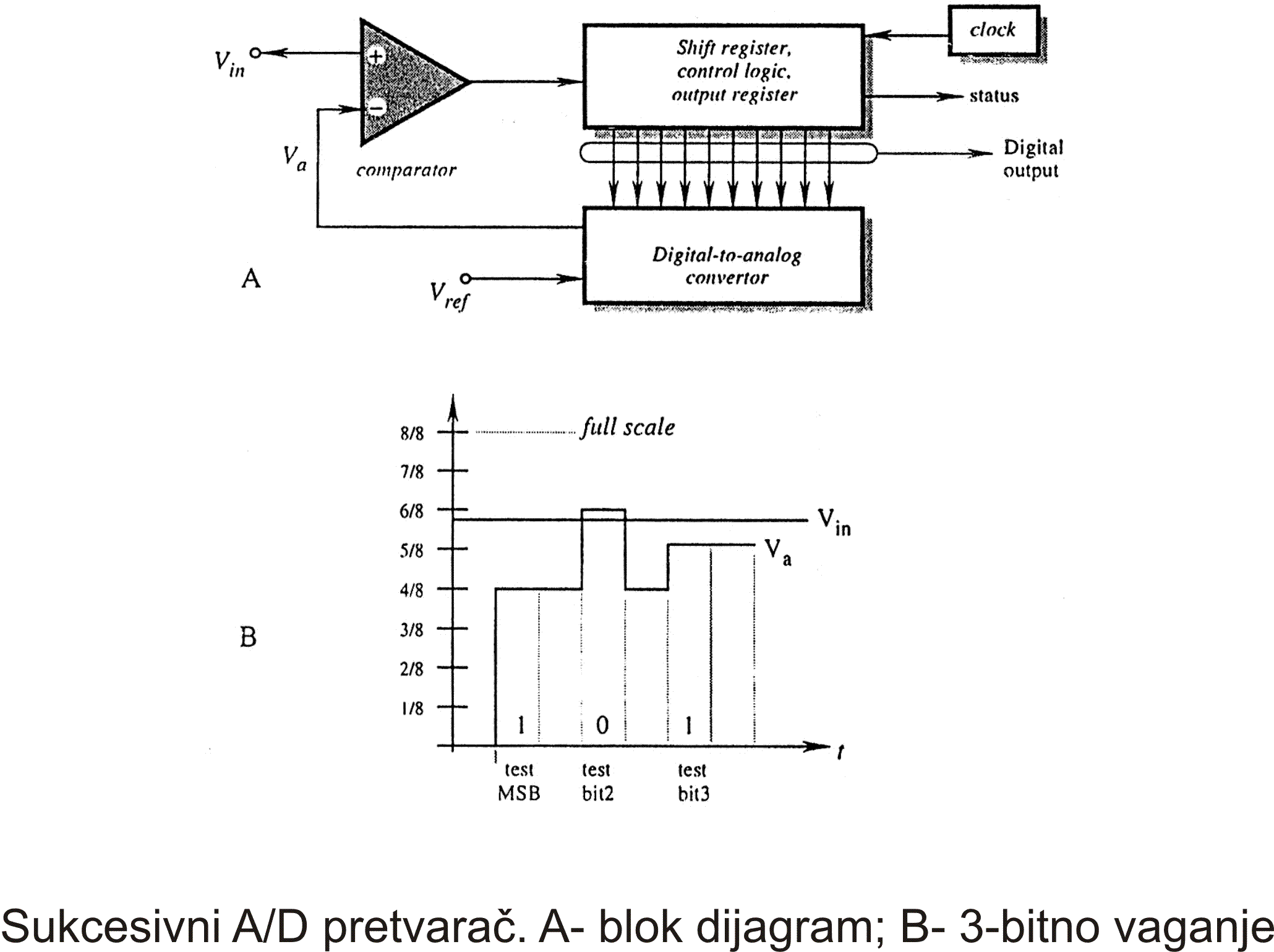
## C programming

* **A void pointer** is a pointer to data that is of a yet to be determined data type
* It can be cast to point to any data object you would like .
* It is useful in passing parameters of different variable types into a function
* int a=10;
* float b=35.75;
* void \*ptr; // Declaring a void pointer
* ptr=&a; // Assigning address of integer to void pointer.
* printf("The value of integer variable is= %d",\*( (int\*) ptr) );// (int\*)ptr - is used for type casting. Where as \*((int\*)ptr) dereferences the typecasted void pointer variable.
* ptr=&b; // Assigning address of float to void pointer.
* printf("The value of float variable is= %f",\*( (float\*) ptr) );
* **Pointer to functions**
* Since a function is located at an address in program memory you can construct a pointer to the function
* This feature allows you to pass a function into a function based upon need. It also lets you assign a function based upon say an input and then forevermore use that function once the input has been determined
* int DoIt (float a, char b, char c)
* { return a+b+c; }
* int DoMore(float a, char b, char c)
* { return a-b+c; }
* int main(void) {
* int (\*pt2Function)(float, char, char) = 0;
* int d; char a =2,b=1,c=0;
* pt2Function = DoIt; // short form
* d = (\*pt2Function)(a,b,c); //calling the function using a pointer
* pt2Function = &DoMore; // correct assignment using address //operator
* d = (\*pt2Function)(a,b,c);

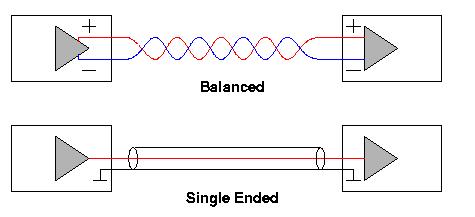
## ADC

* Say a sensor has a source impedance of 1K Ω
* Say the Reading circuit has an impedance of 50K
* Because of the voltage divider a 2% error exists.
* If you use a buffer with say 1 M Ω input impedance and 50 Ohms output impedance than the error goes to .1% + .1% = .2%

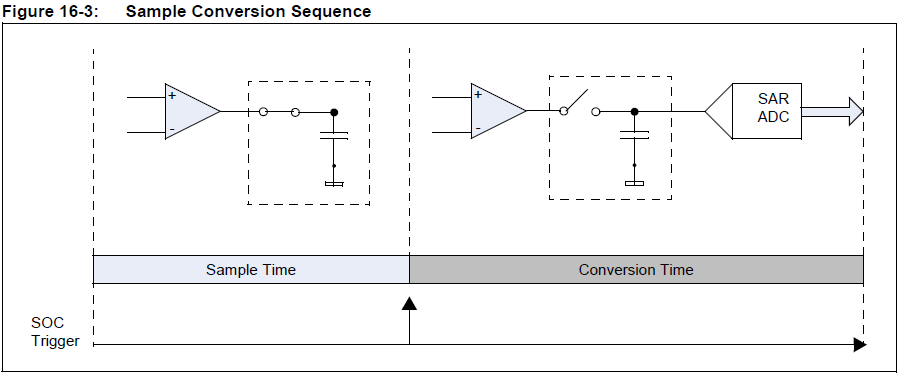


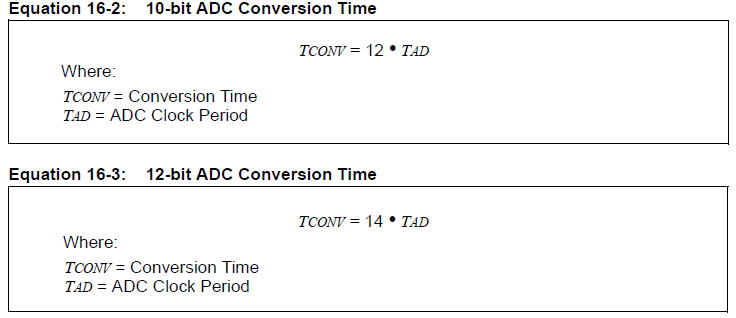


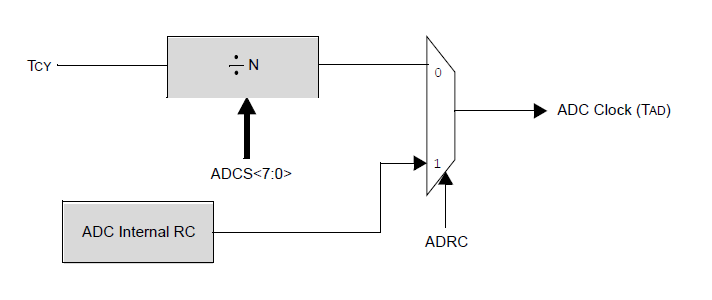
* Min 0 V = Code 0
* Max Vmax = Code 2n – 1
* Resolution = Vmax /(2n – 1)
* Code = D = e/Vmax (2n – 1)

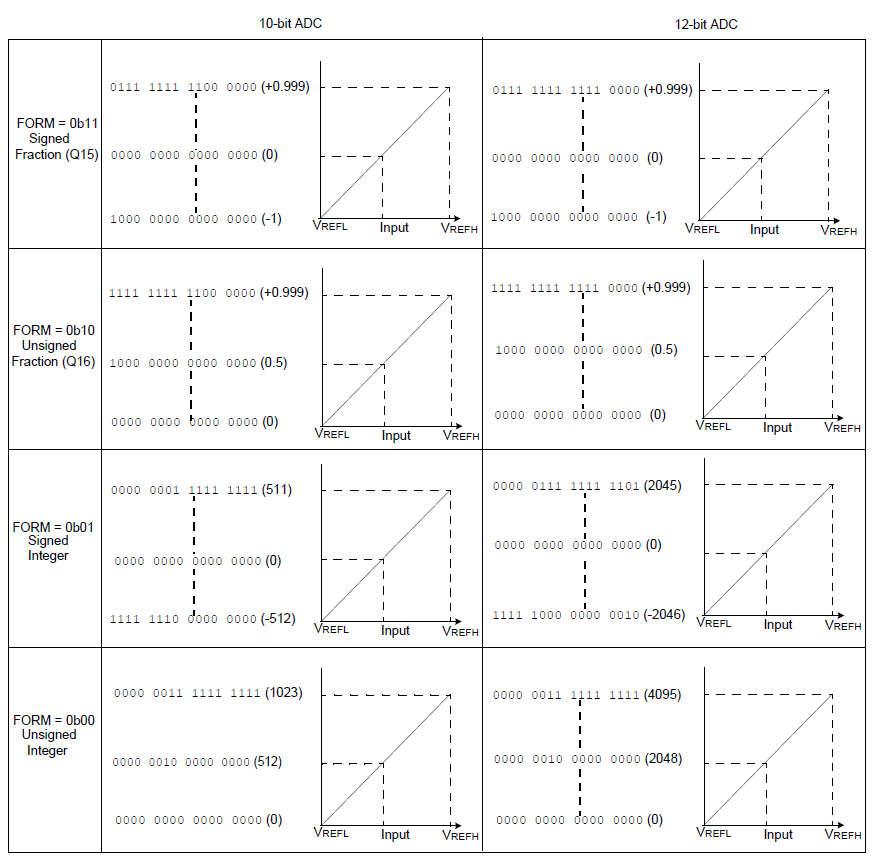


* If the signal you want to read is referenced to a common ground then single ended is possible

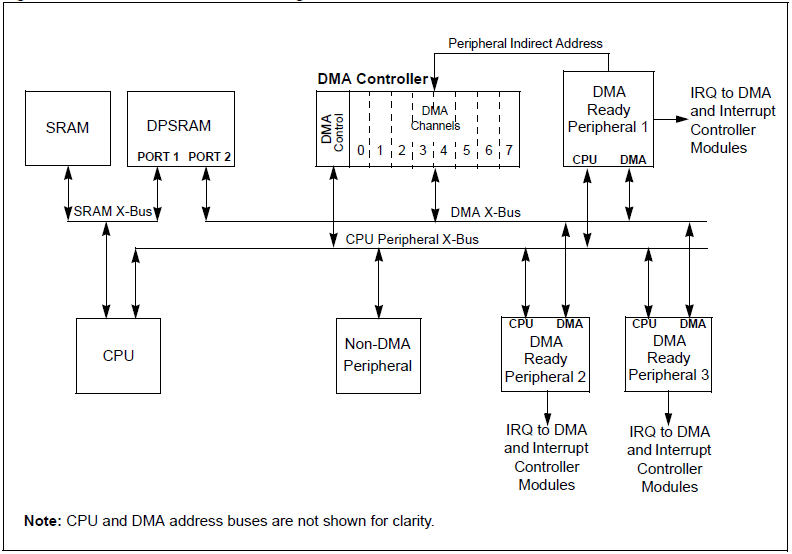








## DMA



**Operating Mode One-Shot or continuous** block transfers

**Ping-Pong mode**

## Power and RMS



* True RMS calculation must average the instantaneous samples therefore in you meter design you will do this calculation at equally sampled points:
*  Apparent Power = Pa=Veff \* Ieff
* 
* Since 60 Hz has a smaller period we will increase the integer number of periods (N) of a 60 Hz wave form until a 50 Hz period will evenly divide into the period (M)
* 
* Therefore the slowest we can sample is: 
* This rate goes into the sampling period of .1 seconds an even 60 times.
* 
* 