|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Properties and Indexer:** | | **Delegates:** 🡪 ähnlich wie „function pointers“ in C | | |
| **GUI-Begriffe 🡪 Form**: selbständiges, vollwertiges verschiebbares Fenster (System.Windows.Forms) **Control:** von .Net definierte GUI-Komponente (müssen in eine andere Control z.B. Panel/Form platziert werden **UserControl:** Basisklasse für selbst generierte GUI-Komponenten **Controls:** Variable innerhalb eines Control, mit der eine Lister der eingebetteten Komponenten vom Type Control verwaltet werden. Nur in der Liste enthaltene Komponenten sind sichtbar. **Size/ClientSize:** Grösse mit/ohne Rand u. Titelleiste. Menüleiste wird nicht berücksichtigt. **Modal-Eigenschaft:** Im Modal-Modus geöffnete Forms müssen geschlossen werden, bevor Eingaben für andere, darunterliegende Fenster möglich sind (Dialogfenster). | |
| **Events (Handling):** |  | | **API Entwicklung für Spezialgeräte**  Die **RobotCtrl-Library** kontrollliert den Robot (Motor, Switches, LED’s,…) | |
|  | |
| **GUI-Programmierung 1**. Eine **leere Solution** wird erstellt. **2.** Ein **„Windows – Form“ Projekt** wird der Solution hinzugefügt **3.** in der Klasse „Form1.cs“ wird nun ein **leeres GUI – Fenster** angezeigt. **4.** Mittels der **Toolbox** können nun verschiedene Elemente mittels Drag – and – Drop ins Fenster gezogen werden. **5.** Diese **Elemente** können nun mithilfe der **Properties** nach Wunsch angepasst werden. **6.** Wird auf ein Element doppelgeklickt, öffnet sich automatisch die Controller – Klasse und ein **Event – Handler** wird erstellt, in welchem implementiert werden kann, was bei dieser Aktion geschehen soll | | | |  |

|  |  |
| --- | --- |
| **Parameters**    The **out** keyword causes arguments to be passed by reference. It is like the **ref** keyword, except that **ref** requires that the variable be initialized before it is passed. To use an **out** parameter, both the method definition and the calling method must explicitly use the **out** keyword. | **Bitmaskierung Einlesen und Schreiben an Ports** |
|  |
| **Motorenansteuerung** Robot besitzt zwei IC LM629 über diese wird v und a eingestellt und Distanz gelesen. Zur Ansteuerung wird **MotorCtrl** erstellt, davon zwei Instanzen für Motor (l,r) erzeugt. **DriveCtrl** Klasse ist dafür zuständig H-Bridge ein-/auszuschalten. |
| **Robot-Fahrsteuerung Ziel:** Beide Motoren separat ansteuern (unabhängig, gleichzeitig und mit Hilfe Prozess-Steuerung -> überprüft stetig den Zustand des Prozesses und macht Anpassungen) **Steuerungsloop:** (Endlosschleife oder mit Timer) 🡪 Bei Endlosschleife minimales Sleep benötigt, damit übrige Softwareteile auch CPU-Leistung erhalten. Einzelne Durchläufe nicht gleiche Dauer.   1. Prozess-Abbild: Beinhaltet alle messbaren und beobachtbaren Sensor Daten 2. Berechnungen/Entscheidungen passieren auf Basis von Punkt 1 3. Neue Prozess-Parameter sind am Zyklusende gesamthaft und gleichzeitig zu setzen bzw. zu aktivieren.   **Common Run Parameters:** UserControl zur Form hinzufügen, Events SpeedChanged und AccelerationChanged abonnieren. Im EventHandler wird v und a in den TrackLine, TrackTurn und TrackArcRight/Left-UserControls gesetzt. |

|  |  |
| --- | --- |
| **Multithreading**  **New** 🡪Das Thread-Objekt ist erzeugt, aber noch nicht gestartet **Ready** 🡪Der Thread ist gestartet, lokal Speicher (stack) ist zugeteilt, er wartet nur noch auf die Zuweisung des Prozessors. **Running** 🡪 Der Thread führt seine Anweisungen auf dem Prozessor aus  **Stopped** 🡪 Der Thread existiert nicht mehr. Das Thread – Objekt jedoch schon und kann, falls referenziert, benutzt werden. Sonst 🡪 Garbage Collector  **Abort (requested)** 🡪 Ein geblockter Thread kann wieder in den ready- Zustand versetzt werden. **Blocked** 🡪 Der Thread muss warten, bis eine Bedingung erfüllt wird z.B **1.** Warten eines Timeouts oder auf das Ende eines anderen Threads **2.** Aufgerufene Betriebssystemroutine muss beendet werden (geschieht beispielsweise bei File-Operationen) **3.** Exception (z.B. von nicht vorhandenen File lesen) **Objects lock-pool (Jedes Objekt hat genau einen Object lock pool)🡪** C# bietet die Möglichkeit von Monitoren und Locks. Wenn ein Codeblock gelockt ist, kann nur ein Thread darin arbeiten. Will ein Thread darauf zugreifen, muss er warten, bis ein Lock wieder freigegeben wird. Die freien Locks werden im lock-pool gespeichert. **Objects wait-pool** 🡪 Will ein Thread auf einen gelockten Codeblock zugreifen, wird er zuerst in den wait-pool verschoben. Sobald ein Lock frei wird, kann dieses im lock-pool „gewonnen“ werden (es gibt keine Reihenfolge, welcher Thread das Lock erhält) und somit auf den geschützten Codebereich zugreifen. Der wait-pool ist somit quasi das Wartezimmer für den Zugriff auf gelockte Code-Segmente. Objects lock-pool und Objects wait-pool müssen zum gleichen Objekt gehören. -> sonst Laufzeitfehler (SynchLockException nur Abk.) |  |
| **Bedarf an Synchronisation bei Nebenläufigkeit** (Wenn parallel arbeitende Programmteile auf eine gemeinsame Ressource zurgreifen wollen) **Unterschied Semaphor und Mutex** (der Zähler Mutex kann nur die Werte 0,1 annehmen und Semaphore nicht-negative Werte) **Speziell an Sempaphor und Mutex** (sind geschützte Instanzen(Zähler) deren Operationen atomar ausgeführt werden) **Einsatz** (Mutex: Schreibender Zugriff auf gemeinesame Ressourcen, Datenbank Speicher, I/O Einheiten; Semaphor: Lesender Zugriff auf gemeinsame Ressourcen die begrenzt sind. Produzenten/Konsumenten Aufgaben, Bounded Buffer, Message-Passing, Channels) **Prozess** (Zugriff auf eigene Ressource -> Speicher, Prozessoren, I/O) **Thread** (teilt sich Ressourcen mit anderen Threads) **Sicherheit bei nebenläufigen Programmen** (keine Verklemmung (Deadlock) durch gegenseitige Zugriffe (Interferenzen) in kritischen Bereichen) **Lebendigkeit** (Kein Livelock, jeder Programmteil erhält eine faire Chance ausgeführt zu werden. -> unconditionaly fair, weak fair and strong fair. |
| (1)Welches werden die Prozesse der Mehrprozess Lösung sein? (2)Wo sind Programmänderungen/Erweiterungen zu machen?   |  | | --- | | Klasse RaceHorse und Turf weden zu prozessen | | Klasse RaceHorse mit main-Methode ergänzen |   Wo/Wie stellen Sie die prozessübergreifende Synchronisation dar?   |  | | --- | | Das EventWaitHadle in der Klasse Latch ändern und Name eines systemweiten Synchronisationsereignisses definieren.  *EventWaitHandle signal = new EventWaitHandle(false, EventResetMode.ManualReset, «hslu.prgsy.latch»);* |   Der Start der Rennpferd-Threads war ein Problem, deshalb wurde Thread.Sleep(500); in Zeile 44 eingefügt. Wie wird das Problem in der Mehrprozess Lösung behoben?   |  | | --- | | Warten durch Console.ReadLine(); statt Sleep  Ein Array von EventWaitHandler. Jedes RaceHorse Objekt erhält ein Array-Element und ruft in der Latch Methode Acquire(Erwerb) das set des EventWaitHandlers auf. Turf wartet mit WaitAll auf alle Array-Elemente. |   Ist das Rennen nun gerechter, als bei der Einzelprozess Lösung?   |  | | --- | | Nein! Es gelten die gleichen Bedingungen wie bei der Thread Lösung. Das EventWaitHandle der Klasse Latch besitzt einen Object-Wait und Lock-Pool durhc die alle Signalisation geht. | |
| Welche Einschränkungen besitzt die Klasse SimpleCheckDate?   |  | | --- | | EchoSever ist blockierend, d.h. er kann nur immer einen Client zur gleichen Zeit bearbeiten. *TcPClient client = listen.AcceptTcpClient();*  EchoServer ist nicht skalierbar, nicht stabil und stützt bei einem Fehler ab. (Behandelt nur TCP-Sockets) Es können nur C# Objekte verarbeitet werden. |   Mit welchem Konstrukt könnte eine Einschränkung von SimpleCheckDate behoben werden?   |  | | --- | | Handler (blockierend), Executor (skalierbar) |   Unhandled Exception: System.Net.Sockets.SocketException: Normalerweise darf jede Socketadresse (Protokoll, Netzwerkadresse oder Anschluss) nur jeweils einmal verwendet werden.   |  | | --- | | Der Port ist bereits druch einen andern Server belegt. |   Wie und mit welchen Parametern starten Sie das Telnet?   |  | | --- | | telnet IP(oder localhost) 4711, im telnet mit «open» beginnnen |   Was müssen Sie im Telnet tun, damit Sie auch eine Antwort erhalten?   |  | | --- | | Irgendwas eingeben + CR (Enter) -> man erhält immer eine Antwort |   Wenn der Client ein Web Browser ist und Kontakt mit SimpleCheckDate aufnimmt. Wie sieht die Anfrage aus, was passiert dann und warum?   |  | | --- | | <http://localhost:4711/12.3.04>, String wird nicht als gültiges Datum erkannt. http Kommando wird als request Zeile eingelesen. |   Wenn Sie einen Client für den SimpleCheckDate Server schreiben müssen. Wie muss die Socket Verbindung auf der Client-Seite erzeugt werden?   |  | | --- | | new TcpClient(host, port); | |

|  |  |
| --- | --- |
| **Threads: 1.1/1.2**  **- CLR = Common Language Runtime** 🡪 ist der Name für eine Laufzeitumgebung, in der .NET-Anwendungen ausgeführt werden  **- parallele Prozesse** (laufen zur selben Zeit auf mehreren Prozessorkerne) und **nebenläufige Prozesse** (wechseln die Prozessorkerne untereinander ab und sind somi „quasi-parallel“)  **- Erzeugen und Starten von Threads** | **2.1 Einfache Synchronisation**  **Einfache Blockierung** 🡪 Zugriff auf gemeinsame Resources sum  **1. busy waiting** (**-** braucht viel CPU-Leistung **+** funktioniert auf jedem Rechner **2.** Bringt aktuellen Thread (main) zum Schlafen (**-** funktioniert allenfalls nicht auf verschiedenen Plattformen, weil Zeit nur für eine Plattform bestimmt wurde **+** braucht weniger CPU Ressourcen) **3.** Der aufrufende Thread (main) blockiert mit **t.Join()** bis der t-Thread beendet ist und fährt danach zeitnah weiter mit der Ausgabe  **lock-Konstrukt** 🡪 Die Threads reservieren einen Codebereich mit Schlüsselwort lock für sich 🡪 Allerdings können mit dem lock-Konstrukt nur Codeblöcke geschützt werden und nicht ganze Methoden. Ein **Monitor** ist die Kapselung eines kritischen Bereichs mit Hilfe einer automatisch verwalteten Sperre. Diese Sperre wird beim Betreten des Monitors gesetzt und beim Verlassen wieder zurückgenommen. Ist die Sperre beim Eintritt in den Monitor bereits von einem anderen Thread gesetzt, muss der aktuelle Thread warten, bis der Konkurrent die Sperre freigegeben hat und den Monitor somit verlassen hat. |
| **2.2 Synchronisierungsereignisse**  **Wait Handle** e.g. *private static EventWaitHandle wh = newAutoResetEvent(false);*  🡪 ***wh.Set();*** //Setzt ein Event aktiv 🡪***wh.WaitOne();*** //Wartet auf dieser Codezeile auf einen aktiven Event --> Thread geht auf dieser Zeile in den Objects wait-pool  🡪 ***wh.Reset();*** //Setzt den Event auf inaktiv  🡪 zwei Arten von Synchronisierungsereignisse 🡪 **AutoResetEvent** und **ManualResetEvent.** Der einzige Unterschied zwischen den beiden besteht darin, dass **AutoResetEvent** automatisch von signalisiert(aktiv) zu nicht signalisiert (inaktiv) geändert wird (muss kein Reset gemacht werden), wenn das Ereignis einen Thread aktiviert. Umgekehrt ist es mit **ManualResetEvent** möglich, eine beliebige Anzahl von Threads über den signalisierten Zustand zu aktivieren, und das Ereignis wird nur in den nicht signalisierten Zustand zurückgesetzt, wenn seine Reset-Methode aufgerufen wird.  **Beispiel eines Latches** (mit Wait Handle) 🡪 Latches sperren so lange, bis sie einmal ausgelöst werden und danach sind sie frei passierbar  **Wait and Pulse** 🡪 Die Threads warten an einem Monitorobjekt, bis dieses eine Impuls erhält, um einen (Pulse) oder mehrere (PulseAll) Threads frei zu schalten.  Um einen Thread auf eine Bedinung oder Zustand warten zu lassen stehen folgende Methoden zur Verfügung: *public static bool* ***Wait****(Object obj); public static bool* ***Wait****(Object obj, int millisecondsTimeout); public static bool* ***Wait****(Object obj, TimeSpan timeout)* 🡪 *public static void* ***Pulse****(Object obj); public static void* ***PulseAll****(Object obj)* 🡪 Die Methoden Wait und Pulse werden mit der Klasse Monitor aufgerufen: ***Monitor.Wait****(obj);* ***Monitor.Pulse****(obj);*  **Wichtig:** -Wait/Pulse nur in lock Bereich, Pulse immer erst nach Wait aufrufen  **Ausweg aus Wartezustand**: **1.** Ein anderer Thread signalisiert den Zustandswechsel mittels Pulse bzw. PulseAll. **2.** Die angegebene Zeit (Timeout) ist abgelaufen. **3.** Ein anderer Thread ruft die Methode Abort des wartenden Threads auf. |
| **Semaphore**  Allgemeines Konzept für die Synchronisation. Mutex nur Ja/Nein (lock/unlock), Semaphore definieren die Anzahl Threads die Zugriffe auf einen kritischen Abschnitt haben sollen. Zwei wichtige Operationen: sema.P() und sema.V():  **sema.P():** «Passieren» s>0 then s=s-1; end; **sema.V():** «Freigeben» s=s+1; end;  sema.WaitOne(); // P(), kritischen Bereich betreten  sema.Release(); // V(), kritischen Bereich verlassen  Semaphore sema = new Semaphore(1, 3); //init: 1, max: 3  while (true) {  Console.WriteLine("Thread waits.");  sema.WaitOne();  Console.WriteLine("Thread is in critical section");  Thread.Sleep(1000);//Only 3 threads here at once  sema.Release();  Console.WriteLine("Thread leaves.");}  // Bestehende Semaphore verwenden / resp. neue erzeugen  try {sema = Semaphore.OpenExisting("ximit.ch/");}  catch (WaitHandleCannotBeOpenedException) {sema = new Semaphore(0, 2, "ximit.ch-Semaphore");}  **Mutex / Wechselseitiger Ausschluss:**  Nebenläufige Prozesse, Threads können nicht gleichzeitig auf Daten zugreifen.  **mut.P():** m>0 then m=0; end;  **mut.V():** m<1 then m=1; end; Mutex ist Prozess übergreifend (Sem nicht)!  mutex.WaitOne([timeout]) // P(), auf Mutex warten  mutex.ReleaseMutex() // V(), Mutex freigeben  static Mutex mutex = new Mutex(false, "ximit.ch-Demo");  static void Main(){ //Wait 5 seconds, if another instance is running: exit  if (!mutex.WaitOne(TimeSpan.FromSeconds(5), false)) {  Console.WriteLine("Another instance is running.");  return; //exit} try {  Console.WriteLine("Running - press Enter to exit");  Console.ReadLine();}  finally { mutex.ReleaseMutex(); }} |
|  |  |