МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ УКРАИНЫ

Одесский национальный университет им. И.И.Мечникова

Институт математики, экономики и механики

Ю.Н.Крапивный

Элементы программирования СЧПУ для модели робота-манипулятора (МРМ)

Методическое пособие к курсу

«Проектирование систем числового программного управления»

Специальности

7.05010201 / 8.05010201 – Компьютерные системы и сети

7.04030101 / 8.04030101 – «Прикладная математика»

ОДЕССА - 2014

Крапивный Ю.Н., Элементы программирования СЧПУ для модели робота-манипулятора (МРМ). Методическое пособие к курсу «Проектирование систем с числовым программным управлением». Методическое издание предназначено для студентов 5-6 курсов, обучающихся по специальности 7.05010201 / 8.05010201 – «Компьютерные системы и сети», 7.04030101 / 8.04030101 – «Прикладная математика».

Рецензенты:
д.т.н., профессор Е.В.Малахов

д.т.н., доцент Ю.А.Гунченко

Методическое издание

Составители :

КРАПИВНЫЙ Юрий Николаевич
канд. физ.-мат. наук, доцент

Методическое пособие к курсу
«Проектирование систем с числовым программным управлением».

Рекомендовано к печати Учёным советом ИМЭМ, протокол №3 от 26.12.2013г.

Оглавление

1. Аппаратная архитектура системы ЧПУ

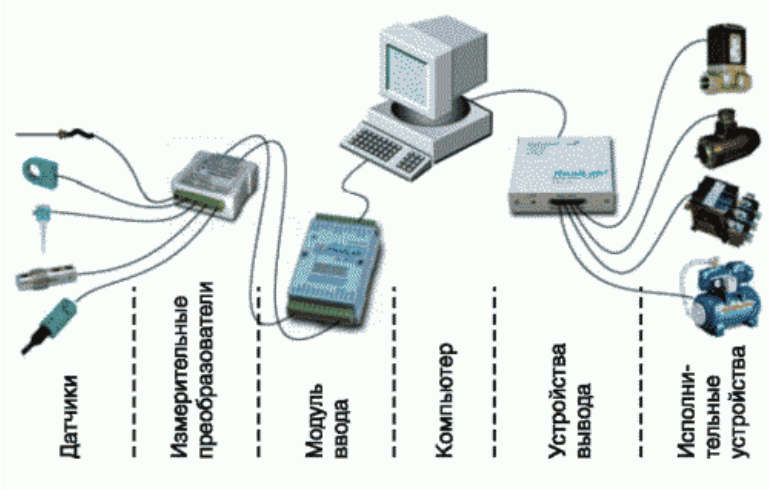
1.1. Архитектура системы

Архитектура электронной части системы ЧПУ проектируется в зависимости от решаемых задач и возможностей УЧПУ. В общем можно сказать, что она состоит из компьютера, специализированных контроллеров, набора необходимых датчиков, устройств ввода-вывода (аналоговых, цифровых), исполнительных устройств (клапанов, реле, двигателей и т.п.) и др.

Для программного управления может использоваться единственный (главный) компьютер, либо несколько компьютеров и программируемых контроллеров, объединённых в сеть. Как правило, в промышленных условиях используются специализированные промышленные сети, такие, например, как:

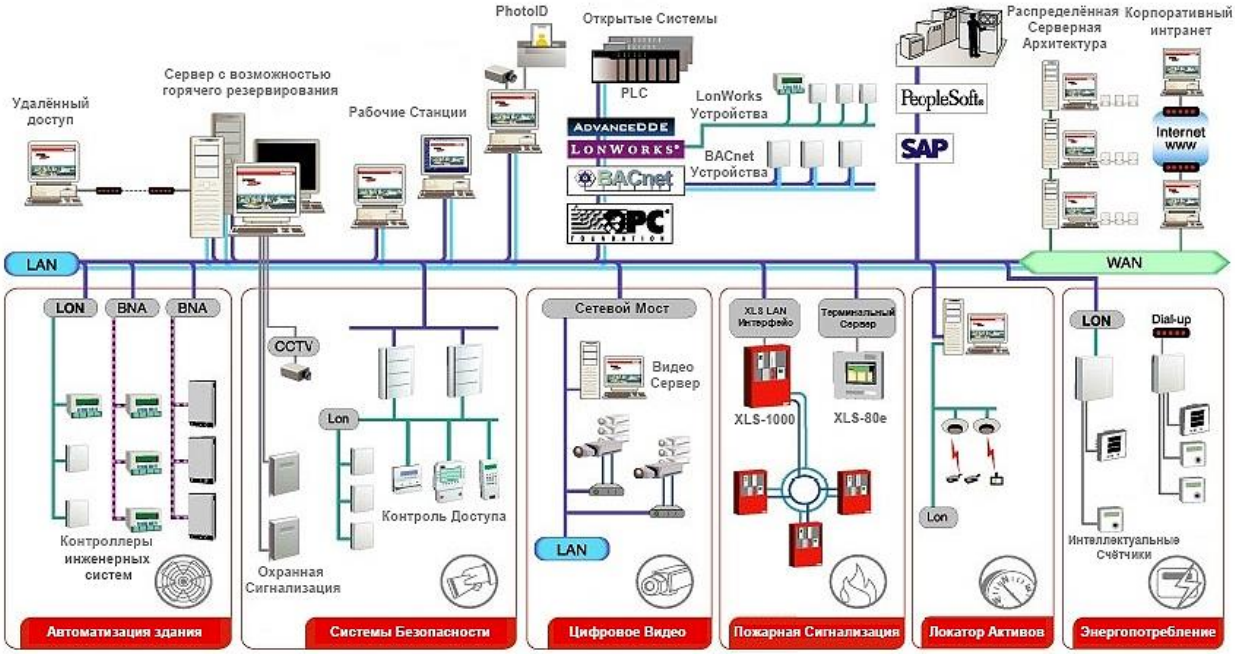
- Сети на основе интерфейсов RS-485 и RS-422: Modbus, Profibus (PROcess FIeld BUS, "промышленная шина для технологических процессов"), DP, ARCNET, BitBus, WorldFip
- CAN (Controller Area Network - "область, охваченная сетью контроллеров"): CANopen, DeviceNet, SDS, J1939
- Сети на основе промышленного Ethernet: Profinet, Foundation Fieldbus HSE (High Speed Ethernet, EtherCAT, Ethernet Powerlink, MODBUS TCP

Архитектура абстрактной системы управления может быть представлена, например, так, как на рисунке [4]:



Ниже приведены примеры архитектур реальных систем управления.

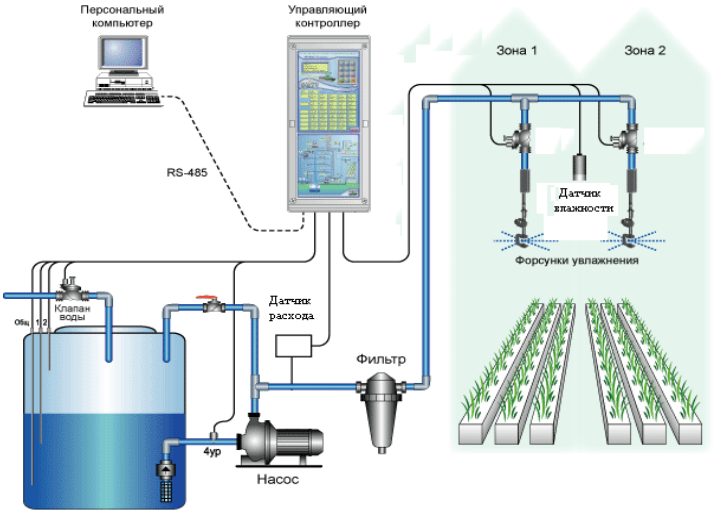
Honeywell Enterprise Buildings Integrator (EBI) [5] - комплексное решение для управления различными системами здания. Разработанная на принципах универсальности и открытой архитектуры, система управления может быть легко усовершенствована и модернизирована. Соответствие системы общепринятым стандартам обмена и обработки данных минимизирует усилия по внедрению её в состав существующих информационных систем.



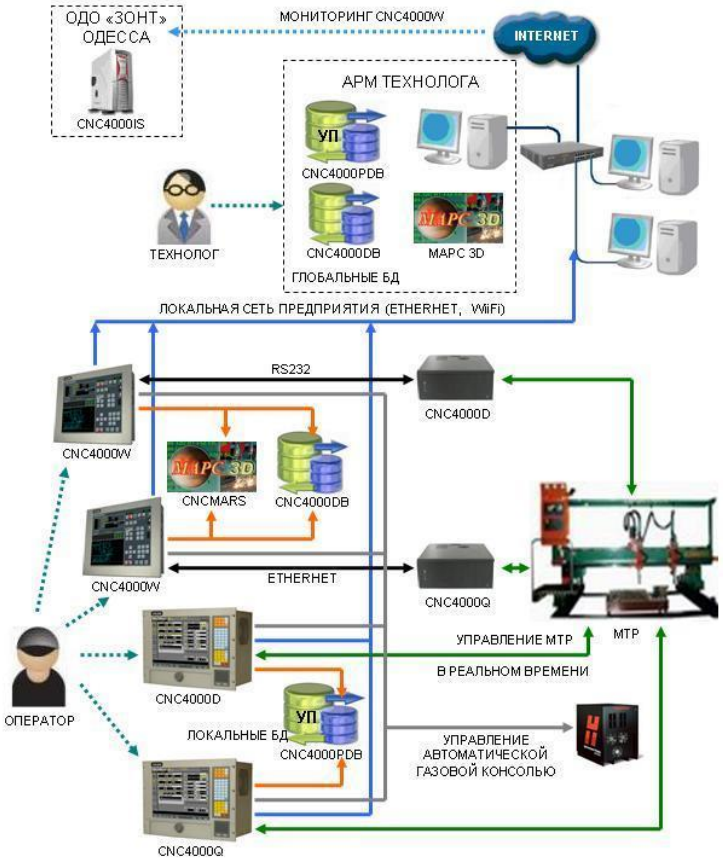
MachineStruxure [6] – комплексное решение для автоматизации машин и механизмов, которое помогает производителям машин и оборудования создавать более эффективное, производительное и надежное оборудование. Ядро архитектуры MachineStruxure – гибкие масштабируемые аппаратные платформы, а также всеобъемлющий пакет программного обеспечения SoMachine.

Пример готовой к использованию и протестированной архитектуры приведён на рисунке

АСУ ККТХ [7] - автоматизированная система управления и контроля климата в тепличных хозяйствах. Архитектура системы имеет два уровня: нижний – подсистема управления (датчики, микроконтроллер, исполнительные механизмы и оборудование) и верхний – пост оператора (персональный компьютер). Связь между уровнями осуществляется по интерфейсу RS-485. Реализация алгоритмов управления осуществляется с помощью автоматизированного модуля верхнего уровня, который также отвечает за интерфейс на посту оператора.



СЧПУ CNC4000 [8] – программно-аппаратный комплекс для управления процессом фигурного раскроя металла на исполнительной машине термической резки. Состоит из ряда модулей, конфигурирование которых позволяет получить конкретную реализацию, ориентированную как на предпочтения заказчиков, так и на оптимальное выполнение требуемых функций.



1.2. Аппаратные компоненты СЧПУ

Для примера рассмотрим аппаратную архитектуру УЧПУ, в которой всё управление обеспечивается единственным мастер-компьютером. В промышленных условиях для этих целей часто используется промышленный компьютер (рабочая станция), возможно уже имеющая и пульт оператора:



Рис.1. Промышленные рабочие станции

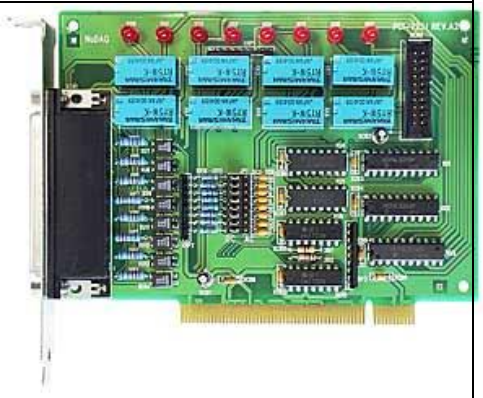
Промышленные компьютеры существенно отличаются от офисных по конструктивным признакам, однако используют те же микропроцессоры и архитектуру. Разъемы для сменных плат устанавливаются на пассивной объединительной панели (кросс-плата), а не на материнской плате.

Собственно, такое понятие, как «материнская плата» отсутствует, так как в один из разъёмов кросс-платы вставляется одноплатный промышленный компьютер.

Рис.2. Кросс-плата

Рис.3. Одноплатный компьютер

Компьютер обладает ограниченными возможностями для управления внешними устройствами. Фактически они сводятся к возможностям обеспечения взаимодействия через порты ввода-вывода, организующее взаимодействие процессора и устройств ввода-вывода. Так организована, например, работа с последовательными (COM) и параллельными (LPT) портами. Для расширения возможностей взаимодействия в кросс-плату промышленного компьютера могут вставляться специализированные платы (например, плата ЦАП), а в архитектуре системы присутствуют специализированные устройства (например, сервоусилители). Примеры плат, устройств и механизмов приведены в таблице.

Назначение устройства	Фотография
ЦАП - цифро-аналоговый преобразователь. Предназначена для преобразования 16-битного цифрового кода в аналоговый сигнал, лежащий в пределах от +10В до -10В.	
МВВ – модуль ввода-вывода. В системах автоматизации очень распространены двоичные сигналы, которые поступают от концевых выключателей, датчиков охранной или пожарной сигнализации, датчиков заполнения емкостей, датчиков сбегания ленты на конвейере, датчиков приближения и т. п. Такие сигналы принято называть "дискретными". Вывод дискретных сигналов используется для управления состоянием включено/выключено исполнительных устройств.	
КИП - контрольно-измерительный преобразователь. Предназначен для измерения точного положения вала двигателя. Положение может считывается с резольверов, фотодатчиков и т.п. установленных, например, на валу двигателя. Данные передаются на общую шину (ISA, PCI). КИП даёт определённое разрешение на оборот двигателя (например, 4096 дискрет/оборот). Может содержать несколько независимых каналов контрольно-измерительного преобразователя.	
Сервопривод - усилитель (привод) координатный, предназначен для управления двигателями постоянного тока. Управляется аналоговым сигналом в диапазоне от +10В до -10В (например, от ЦАП).	

Электродвигатель. Предназначен для обеспечения различных перемещений в системе (линейных, круговых) и т.п. Для аналогового сервопривода применяются двигатели постоянного тока. Как правило, между исполнительным устройством и двигателем устанавливается редуктор, понижающий обороты двигателя и усиливающий крутящий момент, что позволяет создавать роботов, обладающих большой мощностью.	
Датчики: температуры, давления, касания, конечные выключатели и т.п. Формируемый датчиками сигнал может быть дискретным (включён-выключен), аналоговыми (уровень напряжения), цифровыми (цифровой код).	
Электрические клапаны. Предназначены управления подачей жидкостей и газов.	

В качестве примера объекта управления рассмотрим модель промышленного робота-манипулятора. Представление о таком роботе можно получить по рисункам:

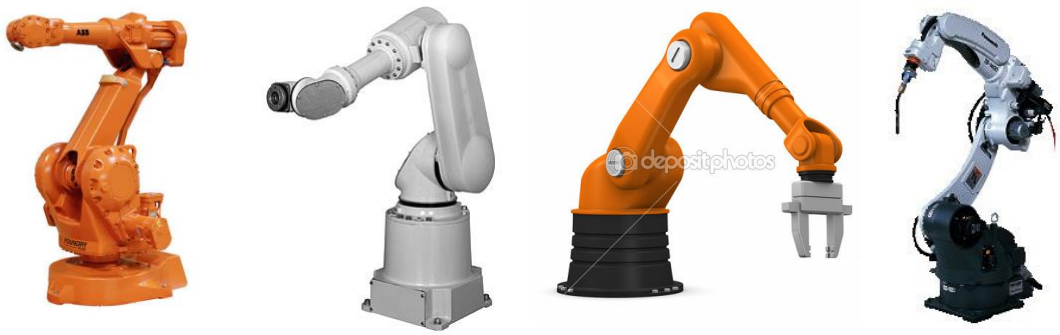


Рис.4. Роботы-манипуляторы

Нетрудно видеть, что роботы могут обладать различным набором степеней свободы, что соответствует независимым координатам. В первую очередь это координаты, обеспечивающие вращение определённой части робота в своей плоскости. Сюда можно ещё добавить и линейные координаты, если основание робота способно перемещаться в плоскости XY. Робот представляет собой модель руки человека, поэтому можно выделять такие его части, как плечо, предплечье, запястье:

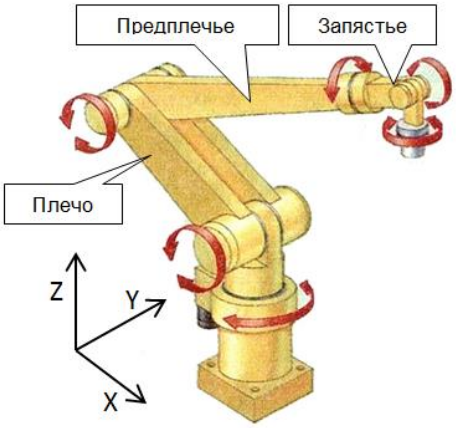


Рис.5. Кинематическая модель робота с 6-тью степенями свободы.

Запястье оснащается инструментом для обеспечения технологических функций робота. Это может быть захват, сверлильная головка, лазерный или плазменный резак, пульверизатор для покраски и т.п.

Наличие достаточного набора степеней свобода обеспечивают запястью робота практически неограниченные точки доступа (положение) в ограниченном пространстве:

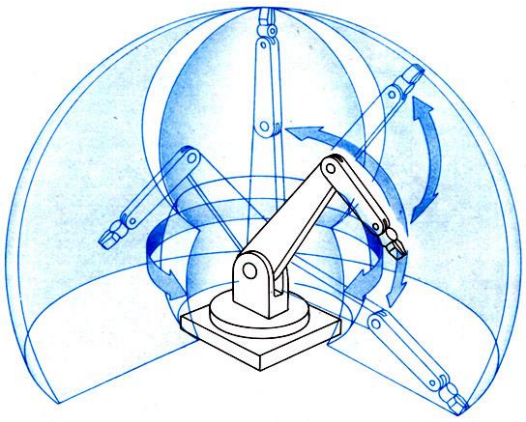


Рис.6. Пространство охвата робота-манипулятора.

1.3. Архитектура модельного робота-манипулятора (МРМ)

Рассмотрим в качестве примера упрощённую МРМ с пятью степенями свободы (координаты X, Y, U_z, U_1, U_2), состоящего из плеча, предплечья с захватом (рис. 7)

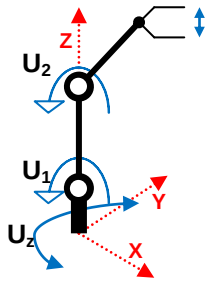


Рис.7. Кинематическая модель РМ с тремя степенями свободы.

Для обеспечения движения по координатам X, Y, U_z, U_1, U_2 каждая из них должна иметь аппаратную архитектуру например такую, как представлено на рис.8.

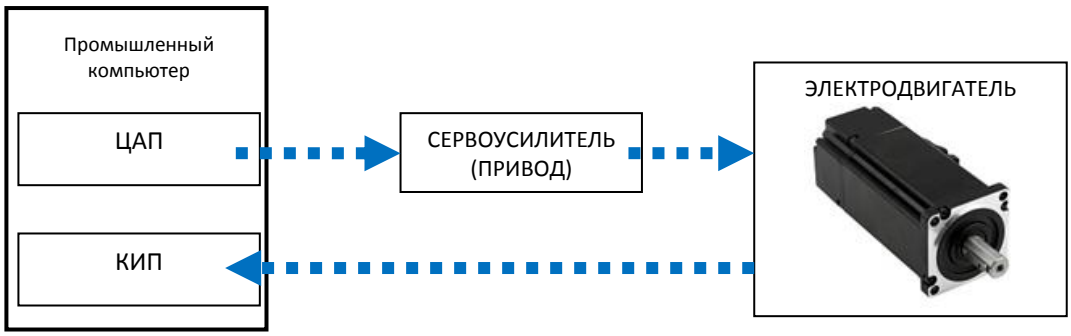


Рис.8. Аппаратная архитектура для одной координаты.

Отметим, что здесь рассматривается одна из возможных аппаратных архитектур для координаты. В случае, например, использования интеллектуальных приводов, включающих в себя контроллер движения, архитектура будет иной.

Система управления координатой по сути представляет собой систему автоматического регулирования с обратной связью. Анализ состояния системы и формирование управляющего воздействия выполняется с заданной периодичностью (тактом) с требованиями жёсткого реального времени. Период (такт) управления выбирается из расчёта обеспечения заданных характеристик системы: точность позиционирования, скорость отработки, устойчивости и т.п. Способ определения длительности здесь не рассматривается, однако можно сказать, что чем период меньше, тем более высокого

качества управления можно добиться. Для определённости будем считать длину такта T=1мс (0.001сек.).

Упрощённая технология управления координатой для обеспечения точного позиционирования может быть представлена следующим алгоритмом:

1. В начале каждого такта T модуль интерполяции ПО УЧПУ производит расчёт заданного положения для координаты на момент окончания очередного такта. При этом учитывается общая траектория движения в пространстве по всем координатам, заданная скорость перемещения, заданное ускорение и т.п.
2. Через КИП считывается реальная позиция координаты в дискретах (импульсах) датчика обратной связи. Вычисляется ошибка между реальным (достигнутым) и заданным (которое нужно было достичь исходя из задания предыдущего такта) положением на начало текущего такта.
3. С учётом заданного положения (см. шаг 1) и ошибки (см. шаг 2) формируется цифровой код задания на отработку в текущем такте. Задание может быть сформировано, например, с использованием ПИД-регулятора (см. далее).
4. Код передаётся в ЦАП, где преобразовывается в соответствующее напряжение (аналоговый сигнал).
5. Аналоговый сигнал с ЦАП передаётся в сервопривод, усиливается сервоприводом в соответствии с техническими характеристиками двигателя и напряжение подаётся на двигатель.
6. В соответствии с заданным напряжением вал двигателя выполняет вращение, которое через редуктор передаётся исполнительному механизму. При вращении вала изменяется и положение датчика обратной связи, что позволяет отследить новое положение координаты.
7. Цикл управления повторяется начиная с шага 1.

Код задания на шаге 4, может быть сформирован как для системы автоматического регулирования с обратной связью на основе классического пропорционального интегро-дифференциального (ПИД) регулятора [9]. Простейшая система такого регулирования показана на рис.9. В ней блок ***R*** называют регулятором (от слова Regulator), ***P*** - объектом регулирования (от слова Process), ***r*** - управляющим воздействием или уставкой (reference), ***e*** - *сигналом рассогласования* или *ошибки* (error), ***u*** - выходной величиной регулятора, ***y*** - регулируемой величиной.

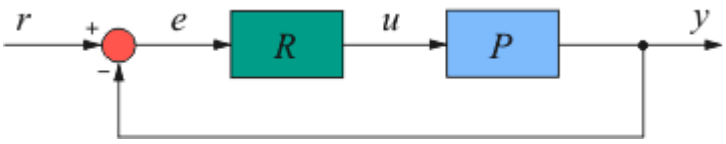


Рис.9. ПИД-регулятор в системе с обратной связью

Регулятор называют ПИД-регулятором, если выходная переменная ***u*** регулятора описывается выражением

$$u(t) = Ke(t) + \frac{1}{T_i}\int_0^t e(t)dt + T_d\frac{de(t)}{dt}$$

где ***t*** - время; ***K*** , T_i , T_d - пропорциональный коэффициент (безразмерный), постоянная интегрирования (размерность времени) и постоянная дифференцирования (размерность времени) регулятора, то такой.

В частном случае пропорциональная, интегральная или дифференциальная компоненты могут отсутствовать и такие упрощенные регуляторы называют П, И или ПИ регуляторами.

2. Многопоточная архитектура приложений

2.1. Введение в многопоточность на С.

По материалам [1] http://www.hardline.ru/1/5/1530/ , [11] http://msdn.microsoft.com/ru-ru/library/ms173178

Введение

В главе рассматриваются методы синхронизации потоков одного или нескольких процессов. Все методы основаны на создании специальных объектов синхронизации. Эти объекты характеризуются состоянием. Различают сигнальное и несигнальное состояние. В зависимости от состояния объекта синхронизации один поток может узнать об изменении состояния других потоков или общих (разделяемых) ресурсов.

Небольшое замечание: функция _beginthread, используемая в примерах, может быть заменена соответствующим эквивалентом MFC (AfxBeginThread) или аналогичной в других диалектах языка С.

Несинхронизированные потоки

Первый пример иллюстрирует работу с несинхронизированными потоками. Основной цикл, который является основным потоком процесса, выводит на экран содержимое глобального массива целых чисел. Поток, названный "Thread", непрерывно заполняет глобальный массив целых чисел.

```
#include <process.h>
#include <stdio.h>
  int a[ 5 ];
  void Thread( void* pParams )
{ int i, num = 0;
    while ( 1 )
  {
     for ( i = 0; i < 5; i++ ) a[ i ] = num;
     num++;
  }
}
 int main( void )
{
   _beginthread( Thread, 0, NULL );
     while( 1 )
      printf("%d %d %d %d %d\n",
             a[ 0 ], a[ 1 ], a[ 2 ],
             a[ 3 ], a[ 4 ] );
   return 0;
}
```

Как видно из результата работы процесса, основной поток (сама программа) и поток Thread действительно работают параллельно (красным цветом обозначено состояние, когда основной поток выводит массив во время его заполнения потоком Thread):

```
81751652 81751652 81751651 81751651 81751651
81751652 81751652 81751651 81751651 81751651
83348630 83348630 83348630 83348629 83348629
83348630 83348630 83348630 83348629 83348629
83348630 83348630 83348630 83348629 83348629
```

Запустите программу, затем нажмите "Pause" для остановки вывода на дисплей (т.е. приостанавливаются операции ввода/вывода основного потока, но поток Thread продолжает свое выполнение в фоновом режиме) и любую другую клавишу для возобновления выполнения.

Критические секции

А что делать, если основной поток должен читать данные из массива после его обработки в параллельном процессе? Одно из решений этой проблемы - использование критических секций.

Критические секции обеспечивают синхронизацию подобно мьютексам (о мьютексах см. далее) за исключением того, что объекты, представляющие критические секции, доступны в пределах одного процесса. События, мьютексы и семафоры также можно использовать в "однопроцессном" приложении, однако критические секции обеспечивают более быстрый и более эффективный механизм взаимно-исключающей синхронизации. Подобно мьютексам объект, представляющий критическую секцию, может использоваться только одним потоком в данный момент времени, что делает их крайне полезными при разграничении доступа к общим ресурсам. Трудно предположить что-нибудь о порядке, в котором потоки будут получать доступ к ресурсу, можно сказать лишь, что система будет "справедлива" ко всем потокам.

```
#include <windows.h>
#include <process.h>
#include <stdio.h>
  CRITICAL_SECTION cs;
int a[ 5 ];
  void Thread( void* pParams )
{
  int i, num = 0;
    while ( TRUE )
  {
     EnterCriticalSection( &cs );
     for ( i = 0; i < 5; i++ ) a[ i ] = num;
     LeaveCriticalSection( &cs );
     num++;
  }
}
  int main( void )
{
  InitializeCriticalSection( &cs );
  _beginthread( Thread, 0, NULL );
  while( TRUE )
  {
     EnterCriticalSection( &cs );
     printf( "%d %d %d %d %d\n",
             a[ 0 ], a[ 1 ], a[ 2 ],
             a[ 3 ], a[ 4 ] );
     LeaveCriticalSection( &cs );
  }
  return 0;
}
```

Мьютексы (взаимоисключения)

Мьютекс (англ. mutex, от mutual exclusion — «взаимное исключение») - это объект синхронизации, который устанавливается в особое сигнальное состояние, когда не занят каким-либо потоком. Только один поток владеет этим объектом в любой момент времени, отсюда и название таких объектов - одновременный доступ к общему ресурсу исключается. Например, чтобы исключить запись двух потоков в общий участок памяти в одно и то же время, каждый поток ожидает, когда освободится мьютекс, становится его владельцем и только потом пишет что-либо в этот участок памяти. После всех необходимых действий мьютекс освобождается, предоставляя другим потокам доступ к общему ресурсу.

Два (или более) процесса могут создать мьютекс с одним и тем же именем, вызвав метод CreateMutex . Первый процесс действительно создает мьютекс, а следующие процессы получают хэндл уже существующего объекта. Это дает возможность нескольким процессам получить хэндл одного и того же мьютекса, освобождая программиста от необходимости заботиться о том, кто в действительности создает мьютекс. Если используется такой подход, желательно установить флаг bInitialOwner в FALSE, иначе возникнут определенные трудности при определении действительного создателя мьютекса.

Несколько процессов могут получить хэндл одного и того же мьютекса, что делает возможным взаимодействие между процессами. Вы можете использовать следующие механизмы такого подхода:

Дочерний процесс, созданный при помощи функции CreateProcess может наследовать хэндл мьютекса в случае, если при его (мьютекса) создании функией CreateMutex был указан параметр lpMutexAttributes .

Процесс может получить дубликат существующего мьютекса с помощью функции DuplicateHandle .

Процесс может указать имя существующего мьютекса при вызове функций OpenMutex или CreateMutex .

Вообще говоря, если вы синхронизируете потоки одного процесса, более эффективным подходом является использование критических секций.

```
#include <windows.h>
#include <process.h>
#include <stdio.h>
  HANDLE hMutex;
int a[ 5 ];
  void Thread( void* pParams )
{
   int i, num = 0;
     while ( TRUE )
   {
      WaitForSingleObject( hMutex, INFINITE );
      for ( i = 0; i < 5; i++ ) a[ i ] = num;
      ReleaseMutex( hMutex );
      num++;
   }
}
  int main( void )
{
   hMutex = CreateMutex( NULL, FALSE, NULL );
   _beginthread( Thread, 0, NULL );
     while( TRUE )
   {
      WaitForSingleObject( hMutex, INFINITE );
      printf( "%d %d %d %d %d\n",
              a[ 0 ], a[ 1 ], a[ 2 ],
              a[ 3 ], a[ 4 ] );
      ReleaseMutex( hMutex );
   }
   return 0;
}
```

События

А что, если мы хотим, чтобы в предыдущем примере второй поток запускался каждый раз после того, как основной поток закончит печать содержимого массива, т.е. значения двух последующих строк будут отличаться строго на 1?

Событие - это объект синхронизации, состояние которого может быть установлено в сигнальное путем вызова функций SetEvent или PulseEvent . Существует два типа событий:

Тип объекта	Описание
Событие с ручным	Это объект, сигнальное состояние которого сохраняется до ручного сброса функцией

сбросом	ResetEvent . Как только состояние объекта установлено в сигнальное, все находящиеся в цикле ожидания этого объекта потоки продолжают свое выполнение (освобождаются).
Событие с автоматическим сбросом	Объект, сигнальное состояние которого сохраняется до тех пор, пока не будет освобожден единственный поток, после чего система автоматически устанавливает несигнальное состояние события. Если нет потоков, ожидающих этого события, объект остается в сигнальном состоянии.

События полезны в тех случаях, когда необходимо послать сообщение потоку, сообщающее, что произошло определенное событие. Например, при асинхронных операциях ввода и вывода из одного устройства, система устанавливает событие в сигнальное состояние когда заканчивается какая-либо из этих операций. Один поток может использовать несколько различных событий в нескольких перекрывающихся операциях, а затем ожидать прихода сигнала от любого из них.

Поток может использовать функцию CreateEvent для создания объекта события. Создающий событие поток устанавливает его начальное состояние. В создающем потоке можно указать имя события. Потоки других процессов могут получить доступ к этому событию по имени, указав его в функции OpenEvent .

Поток может использовать функцию PulseEvent для установки состояния события в сигнальное и затем сбросить состояние в несигнальное после освобождения соответствующего количества ожидающих потоков. В случае объектов с ручным сбросом освобождаются все ожидающие потоки. В случае объектов с автоматическим сбросом освобождается только единственный поток, даже если этого события ожидают несколько потоков. Если ожидающих потоков нет, PulseEvent просто устанавливает состояние события в несигнальное.

```
#include <windows.h>
#include <process.h>
#include <stdio.h>
HANDLE hEvent1, hEvent2;
int a[ 5 ];
void Thread( void* pParams )
{
   int i, num = 0;
   while ( TRUE )
   {
      WaitForSingleObject( hEvent2, INFINITE );
      for ( i = 0; i < 5; i++ ) a[ i ] = num;
      SetEvent( hEvent1 );
      num++;
   }
}
int main( void )
{
   hEvent1 = CreateEvent( NULL, FALSE, TRUE, NULL );
   hEvent2 = CreateEvent( NULL, FALSE, FALSE, NULL );
   _beginthread( Thread, 0, NULL );
   while( TRUE )
   {
      WaitForSingleObject( hEvent1, INFINITE );
      printf( "%d %d %d %d %d\n",
              a[ 0 ], a[ 1 ], a[ 2 ],
              a[ 3 ], a[ 4 ] );
      SetEvent( hEvent2 );
   }
   return 0;
}
```

2.2. Потоки на C++ в Windows CE

По материалам [2] http://wm-help.net/books-online/book/59464/59464.html

Потоки создаются функцией CreateThread(). Её описание имеет вид

Потоки на C++ в Windows 98/XP/2000/NT
Функция **CreateThread** создает поток, который выполняется в пределах виртуального адресного пространства

вызывающего процесса.

При каждом вызове этой функции система создает объект ядра (поток). Это не сам поток, а компактная структура данных, которая используется операционной системой для управления потоком и хранит статистическую информацию о потоке. Если необходимо система выделяет память под стек потока из адресного пространства процесса. Новый поток выполняется в контексте того же процесса, что и родительский поток. Поэтому он получает доступ ко всем описателям объектов ядра, всей памяти и стекам всех потоков в процессе. За счет этого потоки в рамках одного процесса могут легко взаимодействовать друг с другом.

CreateThread - это Windows-функция, создающая поток. Если вы пишете код на C/C вместо нее используется _beginthread.

Чтобы создавать поток, который запускается в виртуальном адресном пространстве другого процесса, используется функция CreateRemoteThread.

Синтаксис

```
HANDLE CreateThread(
LPSECURITY_ATTRIBUTES lpThreadAttributes, // дескриптор защиты
SIZE_T dwStackSize,                       // начальный размер стека
LPTHREAD_START_ROUTINE lpStartAddress,    // функция потока
LPVOID lpParameter,                       // параметр потока
DWORD dwCreationFlags,                    // опции создания
LPDWORD lpThreadId                        // идентификатор потока

);
```

Параметры

lpThreadAttributes

[in] Указатель на структуру SECURITY_ATTRIBUTES, которая обуславливает, может ли возвращенный дескриптор быть унаследован дочерними процессами. Если lpThreadAttributes является значением ПУСТО (NULL), дескриптор не может быть унаследован.

Windows NT/2000/XP: член структуры lpSecurityDescriptor определяет дескриптор безопасности для нового потока. Если lpThreadAttributes имеет значение ПУСТО (NULL), поток получает заданный по умолчанию дескриптор защиты. Списки контроля доступа (ACL) в заданном по умолчанию дескрипторе безопасности для потока поступают из первичного маркера или маркера заимствования прав создателя.

dwStackSize

[in] Начальный размер стека, в байтах. Система округляет это значение до самой близкой страницы памяти. Если это значение нулевое, новый поток использует по умолчанию размер стека исполняемой программы.

Обратите внимание! на то, что, в случае необходимости, размер стека растет.

lpStartAddress

[in] Указатель на определяемую программой функцию типа LPTHREAD_START_ROUTINE, код которой исполняется потоком и обозначает начальный адрес потока. Для получения дополнительной информации о функции потока, см. ThreadProc.

lpParameter

[in] Указатель на переменную, которая передается в поток.

dwCreationFlags

[in] Флаги, которые управляют созданием потока. Если установлен флаг CREATE_SUSPENDED, создается поток в состоянии ожидания и не запускается до тех пор, пока не будет вызвана функция ResumeThread. Если это значение нулевое, поток запускается немедленно после создания. В настоящее время никакие другие значения не поддерживаются.

Windows XP: Если установлен флажок STACK_SIZE_PARAM_IS_A_RESERVATION, параметр dwStackSize задает начальный резервный размер стека. Иначе, dwStackSize устанавливает фиксированный размер.

lpThreadId

[out] Указатель на переменную, которая принимает идентификатор потока.

Windows NT/2000/XP: Если этот параметр имеет значение ПУСТО (NULL), идентификатор потока не возвращается.

Windows 95/98/Me: Этот параметр не может быть значением ПУСТО (NULL).

Возвращаемые значения

Если функция завершается успешно, величина возвращаемого значения - дескриптор нового потока.

Если функция завершается с ошибкой, величина возвращаемого значения - ПУСТО (NULL).

Потоки на C++ в Windows CE

Синтаксис

```
HANDLE CreateThread(
  LPSECURITY_ATTRIBUTES lpsa,
  DWORD cbStack,
  LPTHREAD_START_ROUTINE lpStartAddr,
  LPVOID lpvThreadParam,
  DWORD fdwCreate,
  LPDWORD lpIDThread
);
```

Параметры

lpsa

[in] Игнорируется. Должно быть NULL.

cbStack

[in] Игнорируется пока флаг STACK_SIZE_PARAM_IS_A_RESERVATION не используется. В противном случае параметр cbStack определяет размер виртуальной памяти, зарезервированный для нового потока.

Когда параметр cbStack игнорируется, размер стека для потока по умолчанию определяется ключём линковки /STACK.

lpStartAddr

[in] Указатель на определяемую программой функцию типа LPTHREAD_START_ROUTINE, код которой исполняется потоком и обозначает начальный адрес потока.

Note:

Не допустимо использовать значение StartAddr=NULL. В этом случае функция завершается ошибкой с возвращаемым значением ERROR_INVALID_PARAMETER.

lpvThreadParam

[in] Указатель на единственный 32-битный параметра переданнающийся в поток.

fdwCreate

[in] Флаги, которые управляют созданием потока. Если установлен флаг CREATE_SUSPENDED, создается поток в состоянии ожидания и не запускается до тех пор, пока не будет вызвана функция ResumeThread. Если это значение нулевое, поток запускается немедленно после создания. Никакие другие значения не поддерживаются.

lpIDThread

[out] указатель на переменную типа DWORD, в которой функция возвращает идентификатор, приписанный системой новому потоку.

Если передаётсяь NULL (обычно так и делается) – это означает, что Вас нетинтересует идентификатор потока.

Возвращаемые значения :

Если функция завершается успешно, величина возвращаемого значения - дескриптор нового потока.

Если функция завершается с ошибкой, величина возвращаемого значения - ПУСТО (NULL).

Пример создания потока для Windows CE:

```
HANDLE g_htIST;
int n;
DWORD WINAPI ThreadIST(LPVOID lpvParam) // Функция потока
 {
     while (1)          // бесконечный цикл
         {
         n=n+1;         // увеличить счётчик на 1
         Sleep(10);     // заснуть на 10 мс
         }

     return 0;
 }

. . .

// код для создание потока в приостановленном состоянии
g_htIST      = CreateThread(
        NULL,
        0,
        LPTHREAD_START_ROUTINE(ThreadIST),
        NULL,
        CREATE_SUSPENDED,
        NULL
                          );
// запуск потока
ResumeThread( g_htIST );
```

2.3. Потоки на C++ в QNX Momentics

По материалам [3] http://www.qnx.com/developers/docs/6.3.2/neutrino/

Потоки создаются функцией pthread_create(). Её описание имеет вид

Потоки на C++ в QNX Momentics

pthread_create() – создать поток.

Синтаксис:

#include <pthread.h>

int pthread_create(

pthread_t* thread,

const pthread_attr_t* attr,

void* (*start_routine)(void*),

void* arg);

Параметры:

thread

NULL или указатель на объект pthread_t где функция может хранить идентификатор потока нового потока.

attr

Указатель на структуру *pthread_attr_t* определяющую атрибуты нового потока. Вместо того, чтобы манипулировать членами этой структуры напрямую используйте функции *pthread_attr_init ()* и *pthread_attr_set_*. Для исключения см. "QNX расширений," ниже.

Если атрибут равен NULL, то используются атрибуты по умолчанию (см. pthread_attr_init ()).

Если вы измените атрибуты в после создания потока, атрибуты потока не изменятся.

start_routine

Новый thread будет выполнять функцию start_routine с прототипом

*void * start_routine(void *);*

передавая ей в качестве аргумента единственный параметр arg. Если требуется передать более одного параметра, они собираются в структуру, и передается адрес этой структуры. Значение, возвращаемое функцией

start_routine не должно указывать на динамический объект данного thread'a.

arg

Аргумент (указатель) передаваемый в процедуру потока *start_routine*.

Библиотека: libc

Описание:

Функция pthread_create () создает новый поток, с атрибутами, указанными в объекте attr атрибута нить. Созданный поток наследует маску сигналов родительского потока, и его набор ожидающих сигналов пуст.

Дополнительные функции:

pthread_attr_init() - инициализация атрибутов потока. Функция создает объект pthread_attr_t tattr по умолчанию

Синтаксис :

```
#include <pthread.h>

int pthread_attr_init( pthread_attr_t *attr );
```

Параметры :

attr

Указатель на структуру *pthread_attr_t structure* которая должна быть инициализирована.

Библиотека : libc

Функция возвращает 0 после успешного завершения. Любое другое значение указывает, что произошла ошибка.

Значения атрибутов по умолчанию:

Атрибут	Значение	Смысл
scope	PTHREAD_SCOPE_PROCESS	Новый поток не ограничен - не присоединен ни к одному процессу
detachstate	PTHREAD_CREATE_JOINABLE	Статус выхода и поток сохраняются после завершения потока
stackaddr	NULL	Новый поток получает адрес стека, выделенного системой
stacksize	1 Мбайт	Новый поток имеет размер стека, определенный системой
inheritsched	PTHREAD_INHERIT_SCHED	Поток наследует приоритет диспетчеризации родительского потока
schedpolicy	SCHED_OTHER	Новый поток использует диспетчеризацию с фиксированными приоритетами. Поток работает, пока не будет прерван потоком с высшим приоритетом или не приостановится

Для считывания и установки новых значений параметров предназначены специальные функции :

pthread_attr_getdetachstate, pthread_attr__setdetachstate

pthread_attr_getguardsize, pthread_attr_setguardsize

pthread_attr_getinheritsched, pthread_attr_setinheritsched

pthread_attr_getschedparam, pthread_attr_setschedparam

pthread_attr_getschedpolicy, pthread_attr_setschedpolicy

Так, поток может быть создан в одном из двух состояний:

Если поток создается отделенным (PTHREAD_CREATE_DETACHED), его ID и другие ресурсы могут многократно использоваться, как только он завершится. Если нет необходимости ожидать в вызывающем потоке завершения нового потока, можно вызвать перед его созданием функцию pthread_attr_setdetachstate().

Если поток создается неотделенным (PTHREAD_CREATE_JOINABLE – по умолчанию), предполагается, что создающий

поток будет ожидать его завершения и выполнять в созданном потоке pthread_join(). Независимо от типа потока процесс не закончится, пока не завершатся все потоки; pthread_attr_setdetachstate() возвращает 0 - после успешного завершения - или любое другое значение - в случае ошибки.

pthread_attr_setdetachstate() – функция устанавливает новое значение атрибута *detachstate* в структуре определяющей атрибуты нового потока.

Синтаксис :

```
#include <pthread.h>
int pthread_attr_setdetachstate(
        pthread_attr_t* attr,
        int detachstate );
```

Параметры :

attr

Указатель на структуру *pthread_attr_t* определяющую атрибуты нового потока..

detachstate

Новое значение атрибута *detachstate* в структуре:

PTHREAD_CREATE_JOINABLE – создать поток в неотделённом состоянии.

PTHREAD_CREATE_DETACHED – создать поток в отделённом состоянии.

Библиотека : libc

Возвращаемые значения :

EOK - Успешное завершение.

EINVAL – недопустимое значение устанавливаемого параметра.

Пример создания потока для QNX Momentics:

```
int n;
pthread_attr_t attr;


void* ComandThread(void*arg)
{
     while (1)          // бесконечный цикл
         {
         n=n+1;         // увеличить счётчик на 1
         uSleep(1000);  // заснуть на 1000 мкс
         }

 }


. . .

// код для создание потока
 pthread_attr_init( &attr );
 pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_DETACHED );
 pthread_create(NULL, &attr, &ComandThread, NULL);
```

3. Работа с таймером. Обработка прерываний таймера

В этом разделе рассматривается программирование обработчика прерывания таймера для систем жёсткого реального времени.

3.1. Таймер на C++ в Windows CE

Для создания таймера и обработки прерываний таймера используются следующие функции (см. таблицу)

Windows 98/XP/2000/NT/CE

CreateEvent() - создание объекта события функцией

```
HANDLE CreateEvent(
        LPSECURITY_ATTRIBUTES lpEventAttributes,  // атрибут защитыBOOL
        bManualReset,                             // тип сброса TRUE – ручной
        BOOL bInitialState,                       // начальное состояние TRUE – сигнальное
        LPCTSTR lpName                            // имя обьекта);
```

Если функция успешна, то вернется дескриптор события. В том случае, если с таким именем событие уже создано, то вернется дескриптор уже созданного события, а GetLastError() вернет код ERROR_ALREADY_EXISTS. Но функция может вернуть и NULL, если объект события создать не удалось совсем.

SetEvent() - меняет состояние на сигнальное (есть событие).

```
BOOL SetEvent(
        HANDLE hEvent          // дескриптор события
    );
```

В случае успеха вернет ненулевое значение.

ResetEvent() - меняет состояние на невыделенное (нет события):

```
BOOL ResetEvent(
        HANDLE hEvent          // дескриптор события
      );
```

timeSetEvent() - запускает указанное событие мультимедиа таймера. Мультимедиа таймер выполняется в своем собственном потоке. После активации события, оно вызывает указанную функцию обратного вызова или установку некоторого события.

```
MMRESULT timeSetEvent (
        UINT uDelay,
        UINT uResolution,
        LPTIMECALLBACK lpTimeProc,
        DWORD_PTR dwUser,
        UINT fuEvent
);
```

Параметры

uDelay - задержка события в миллисекундах. Если это значение лежит вне диапазона допустимых значений задержки, поддерживаемых таймером, функция возвращает ошибку.

uResolution - разрешение событий таймера в миллисекундах. Разрешение увеличивается при уменьшении значений; разрешение, установленное в нуль, показывает, что периодические события будут происходить с наибольшей возможной точностью. Для уменьшения системных издержек, тем не менее, вы должны использовать максимальное значение, соответствующее вашему приложению.

lpTimeProc - указатель на функцию обратного вызова, которая вызывается по истечению одиночного события или

периодически по истечению периодических событий. Если fuEvent определяет флаг TIME_CALLBACK_EVENT_SET или TIME_CALLBACK_EVENT_PULSE , то значение параметра lpTimeProc интерпретируется как дескриптор события. Для любых других значений fuEvent , значение lpTimeProc интерпретируется как указатель на функцию со следующей сигнатурой:

void (CALLBACK)(UINT uTimerID , UINT uMsg , DWORD_PTR dwUser , DWORD_PTR dw1 , DWORD_PTR dw2).

dwUser - определяемые пользователем данные.

fuEvent - тип события таймера. Может принимать одно из следующих значений:

TIME_ONESHOT - Событие происходит один раз, после uDelay миллисекунд.

TIME_PERIODIC - Событие происходит каждые uDelay миллисекунд.

TIME_CALLBACK_FUNCTION - По истечении времени Windows вызывает функцию, определяемую значением параметра lpTimeProc. Поведение по умолчанию.

TIME_CALLBACK_EVENT_SET - По истечении времени Windows вызывает функцию SetEvent для установки события, определяемого значением параметра lpTimeProc. Параметр dwUser игнорируется.

TIME_CALLBACK_EVENT_PULSE - По истечении времени Windows вызывает функцию PulseEvent для срабатывания события, определяемого значением параметра lpTimeProc. Параметр dwUser игнорируется.

TIME_KILL_SYNCHRONOUS - Передача этого флага предотвращает происхождение события после вызова функции timeKillEvent.

Возвращаемые значения

Возвращается идентификатор события таймера в случае успеха, или ошибка в противном случае. Функция возвращает NULL , если ее вызов завершился неудачей, и событие таймера не было создано. (Идентификатор события также передается в функцию обратного вызова).

Комментарии

Каждый вызов timeSetEvent для периодических событий таймера требует соответствующего вызова функции timeKillEvent . Создание события с флагами TIME_KILL_SYNCHRONOUS и TIME_CALLBACK_FUNCTION предотвращает происхождение события после вызова функции timeKillEvent .

Windows 98/XP/2000/NT

SetThreadPriority() устанавливает значение приоритета для заданного потока. Это значение, вместе с классом приоритета процесса потока, обуславливает базовый уровень приоритета потока.

Синтаксис

```
BOOL SetThreadPriority(
    HANDLE hThread,     // дескриптор потока
    int nPriority       // уровень приоритета потока
);
```

Параметры

hThread

[in] Дескриптор потока, значение приоритета которого должно быть установлено.

Windows NT/2000/XP: Дескриптор должен иметь право доступа THREAD_SET_INFORMATION связанное с ним. Для получения дополнительной информации, см. статью Защита потока и права доступа.

nPriority

[in] Значение приоритета для потока. Этот параметр может быть одним из следующих значений:

THREAD_PRIORITY_ABOVE_NORMAL

THREAD_PRIORITY_BELOW_NORMAL

THREAD_PRIORITY_HIGHEST

THREAD_PRIORITY_IDLE

THREAD_PRIORITY_LOWEST

THREAD_PRIORITY_NORMAL

THREAD_PRIORITY_TIME_CRITICAL

Возвращаемые значения

Если функция завершается успешно, величина возвращаемого значения - не ноль.

Если функция завершается с ошибкой, величина возвращаемого значения - ноль. Чтобы получить дополнительные данные об ошибках, вызовите GetLastError.

Windows CE 6.0

CeSetThreadPriority() -This function sets the priority for a real-time thread on a thread-by-thread basis.

Синтаксис

```
BOOL CeSetThreadPriority(
  HANDLE hThread,
  int nPriority
);
```

Параметры

hThread

[in] Дискриптор потока.

nPriority

[in] Уровень приоритета устанавливаемый для потока.

Это значение может быть в интервале от 0 до 255. Значение 0 задаёт наивысший приоритет. Список ниже демонстрирует символические имена для некоторых значений приоритета (параметра nPriority):

CE_THREAD_PRIO_256_TIME_CRITICAL

CE_THREAD_PRIO_256_HIGHEST

CE_THREAD_PRIO_256_ABOVE_NORMAL

CE_THREAD_PRIO_256_NORMAL

CE_THREAD_PRIO_256_BELOW_NORMAL

CE_THREAD_PRIO_256_LOWEST

CE_THREAD_PRIO_256_ABOVE_IDLE

CE_THREAD_PRIO_256_IDLE

Возвращаемые значения

TRUE - успешное завершение.

FALSE - ошибка.

Для получение кода ошибки можно вызвать функцию GetLastError.

Пример обработки прерывания таймера для Windows CE 6.0:

```
HANDLE m_hevInterrupt;
HANDLE g_htIST;
int n;

void CALLBACK TimerProc1(UINT uTimerID, UINT uMsg, DWORD_PTR dwUser, DWORD_PTR dw1,
DWORD_PTR dw2)
{   //  CALLBACK-функция по прерыванию таймера устанавливает событие
   SetEvent(m_hevInterrupt);
return;


}
```

```
DWORD WINAPI ThreadIST(LPVOID lpvParam) // Функция потока для обработки прерывания таймера
 {
 // Создание объекта события
     m_hevInterrupt  = CreateEvent(NULL, FALSE, FALSE, NULL);
        if (m_hevInterrupt  == NULL)
        {
               printf("ERROR 1\n");
               return 0;
        }
 // создание мультимедийного таймера на 1мс
 timeSetEvent(2, 0, (LPTIMECALLBACK)TimerProc1, 0, TIME_PERIODIC);

 // установка приоритета потока жёсткого реального времени
 CeSetThreadPriority(GetCurrentThread(), CE_THREAD_PRIO_256_TIME_CRITICAL );

 while(1) // бесконечный цикл ожидания возникновения события таймера
 {
     if (WaitForSingleObject(m_hevInterrupt, INFINITE) == WAIT_OBJECT_0)
     {
       __asm {  // выдача кода в параллельный порт
              mov dx,0x378
              mov al,n
             out dx, al
            }
       if (n==0){n=0xff;} else {n=0;}; // изменение (инверсия) кода выдачи в порт
     }
 }
     return 0;
 }

. . .

// код для создание потока обработки прерывания таймера в приостановленном состоянии
g_htIST      = CreateThread(
       NULL,
       0,
       LPTHREAD_START_ROUTINE(ThreadIST),
       NULL,
       CREATE_SUSPENDED,
       NULL
                          );
// запуск потока
ResumeThread( g_htIST );
```

В результате работы этого примера в регистре данных параллельного порта будут формироваться жёсткие импульсы с частотой 2мс (500Гц).

3.2. Таймер на C++ в QNX Momentics

Для создания таймера и обработки прерываний таймера используются следующие функции

QNX Momentics

InterruptAttach() - подключения обработчика прерывания к источнику прерывания

Синтаксис :

```
#include <sys/neutrino.h>
int InterruptAttach(
        int intr,
        const struct sigevent * (* handler)(void *, int),
        const void * area,
        int size,
        unsigned flags );
```

Параметры:

intr

Параметр intr определяет, к какому прерыванию подключается обработчик. Передаваемые значения определяются стартовым кодом, который перед запуском QNX/Neutrino, среди прочего, инициализирует

контроллер обработки прерываний

handler

Адрес функции, которую надо вызвать. Как видно из прототипа, функция *handler()* возвращает структуру *struct sigevent* (указывающую на тип события, которое следует сгенерировать) и принимает два параметра. Первый передаваемый параметр - *area*, тот самый, который передается функции InterruptAttach(). Второй параметр, id, - идентификатор прерывания, его также возвращает InterruptAttach(). Он применяется для идентификации прерывания, а также для маскирования, демаскирования, блокировки и деблокировки прерывания. Четвертый параметр InterruptAttach(), size, указывает размер (в байтах) области данных, которая передается в параметре area. И, наконец

area

Указатель на область невыгружаемой памяти для коммуникации в вашей процессе, или NULL, если области связи не нужна.

size

Размер (в байтах) области данных, которая передается в параметре *area* .

flags

параметр, flags, управляет различными дополнительными опциями:

_NTO_INTR_FLAGS_END - Указывает, что данный обработчик должен сработать после всех других обработчиков данного прерывания (если они есть).

_NTO_INTR_FLAGS_PROCESS - Указывает на то, что данный обработчик связан с процессом, а не с потоком. Что из этого вытекает, так это условие автоматического отключения обработчика. Если вы определяете этот флаг, обработчик будет автоматически отключен от источника прерывания при завершении процесса. Если этот флаг не определен, обработчик прерывания будет отключен от источника, когда завершится поток, подключивший его.

_NTO_INTR_FLAGS_TRK_MSK - Указывает, что ядро должно отследить, сколько раз данное прерывание было маскировано. Это приводит к несколько большей загрузке ядра, но это необходимо для корректного демаскирования источника прерываний при завершении потока или процесса

Library: libc

InterruptWait() – ожидание аппаратного прерывания

Синтаксис :

```
#include <sys/neutrino.h>
int InterruptWait(
        int flags,
        const uint64_t * timeout );
```

Параметры:

flags

Должно быть равно 0.

timeout

Должно быть равно NULL. Зарезервировано для будущих версий реализации.

Используйте функцию TimerTimeout(), чтобы определить период прерывания.

Библиотека: libc

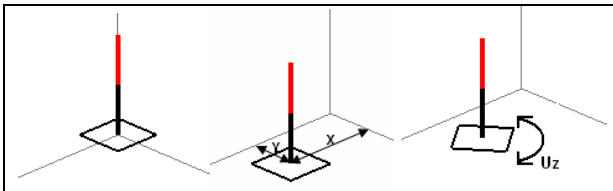
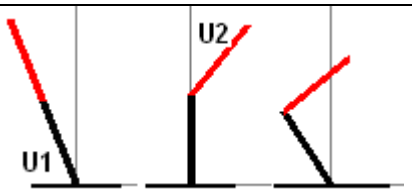
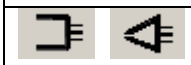
Пример обработки прерывания таймера жёсткого реального времени для QNX Momentics:

```
int n;
int id;
pthread_attr_t attr;
struct sigevent event;

// Обработчик прерываний таймера - генерирует событие, по которому
// запускаетя обработчик dotimtInterrupt() в потоке ComandThread
const struct sigevent *handler( void *area, int id )
{
        return( &event );
}

// функция выполняет некоторые действия по прерыванию таймера
void dotimtInterrupt()
{
```

```
    out8(0x378,n); //выдача кода в параллельный порт

    if (n==0){n=0xff;} else {n=0;}; // изменение (инверсия) кода выдачи в порт

}

void* ComandThread(void*arg) // поток обработки прерывания таймера
{
   // инициализация структуры события
   event.sigev_notify = SIGEV_INTR;

   // подключение потока к вектору прерывания (Attach ISR vector)
   id=InterruptAttach( SYSPAGE_ENTRY(qtime)->intr, &handler, NULL, 0, 0 );

    while (1)  // цикл ожидания возникновения прерывания
    {
          InterruptWait( 0, NULL );    // ожидание возникновения прерывания
                                       // к которому подключился поток
          dotimtInterrupt();           // вызов функции обработки
    }

}


. . .

//директива позволяет потоку иметь доступ к аппаратуре
ThreadCtl(_NTO_TCTL_IO, 0);

// код для создания потока
  pthread_attr_init( &attr );
  pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_DETACHED );
  pthread_create(NULL, &attr, &ComandThread, NULL);
```

4. Программная модель робота-манипулятора (МРМ)

МРМ в программной модели имеет пять степеней свободы [10] – координаты X, Y, Uz, U1, U2 и один захват:

	X, Y – перемещение вдоль соответствующих осей декартовой системы координат Uz – поворот вокруг оси Z
	U1 – наклон первого блока относительно основания U2 – наклон второго блока относительно первого Наклоны производятся в одной плоскости.
	Захват находится на конце второго блока и может находится в двух состояниях – выключен (нет захвата) и включён (есть захват).

Внешний вид окна МРМ приведён в разделе 4.1.

Программная модель МРМ из 3-х динамически загружаемых библиотек (DLL). Все библиотеки МРМ разработаны на языке Delphi (Object Pascal). Интерфейсы экспортируемых функций описаны следующим образом:

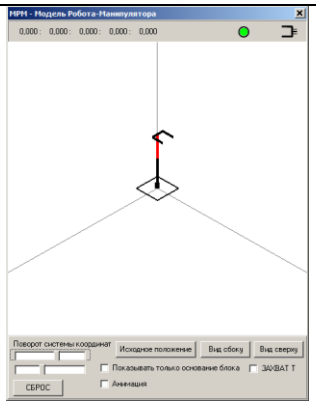
Procedure PName(…);export;stdcall;

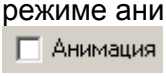
Function FName(…) : ReturnType; export;stdcall;

Далее приведено описание библиотек и функций программного интерфейса МРМ.

4.1. MRM.dll – визуализация МРМ

Динамическая библиотека MRM.dll предназначена для визуализации МРМ. В MRM.dll включены следующие функции:

Интерфейсные функции MRM.dll	
Procedure MRMCreate; - Создание объекта МРМ Создаётся новое окно, содержащее визуализацию МРМ в пространстве:	
Procedure MRMSetON; - Включение МРМ  **Procedure MRMSetOFF;** - Включение МРМ  При выключенной МРМ изменение параметров не воспринимается.	
Процедуры установки пространственных параметров МРМ: **Procedure MRMSetAll(C, U1, U2, X, Y:double);** - установка значений координат C, U1, U2, X, Y **Procedure MRMSetC(C:double);** - установка значений координаты C **Procedure MRMSetU(U1, U2:double);** - установка значений координат U1, U2 **Procedure MRMSetXY(x,y:double);** - установка значений координат X, Y После установки новых значений координат изменяется положение МРМ в окне визуализации в соответствии с новыми значениями.	
Procedure MRMSetM_ON; - включить захват  **Procedure MRMSetM_OFF;** - выключить захват 	

Function MRMBuzy : integer; - возвращает 1, если МРМ занят отработкой задания в режиме анимации, и 0 – в противном случае. Режим анимации включается флагом  на панели визуализации МРМ.

4.2. MRM_IO.dll – эмуляция портов ввода\вывода рабочей станции

Динамическая библиотека MRM.dll предназначена для эмуляции портов ввода\вывода (ПВВ). Состояние ПВВ отображается в файле io.bin, таким образом порты могут быть одновременно доступны нескольким приложениям. Через ПВВ эмулируется работа с внешними устройствами. Каждый порт имеет размер 16 бит (слово). Эмулируется 100 ПВВ единого адресного пространства, начинающегося с некоторого базового адреса (BaseAdr). В MRM_IO.dlll включены следующие функции:

Интерфейсные функции MRM_IO.dll
Procedure IOOpen(BaseAdr:word); - открывает для доступа 100 16-битных ПВВ начиная с базового адреса, заданного параметром BaseAdr.
Procedure IOClose; - закрывает для доступа ПВВ.
Procedure PortOut(PortNo, Value:word); - запись значения Value в ПВВ с адресом PortNo. Значение PortNo должно лежать в интервале [BaseAdr, BaseAdr+99].
Function PortIn(PortNo:word):word; - чтение значения из ПВВ с адресом PortNo. Значение PortNo должно лежать в интервале [BaseAdr, BaseAdr+99].

4.3. MRMPrivod.dll – эмуляция ЦАП, электропривода и ДОС

Динамическая библиотека MRMPrivod.dll предназначена для эмуляции комплекта внешних устройств для управления движением по одной координате (услоное название – ККО – комплект координатного оборудования). В состав комплекта входит:

- ЦАП – цифро-аналоговый преобразователь
- ЭлектроПривод
- Двигатель
- ДОС – датчик обратной связи
- КИП – контрольно-измерительный преобразователь, контроллер обработки ДОС с накопителем

Взаимодействие с комплектом ККО осуществляется через два порта ввода\вывода

- **CAPPort** – порт ЦАП для записи кода задания (напряжения в миливольтах). В реальных системах соответствие записываемого кода в ЦАП и выдаваемого на выходе ЦАПа напряжения зависит от многих факторов, например – от разрядности входного регистра ЦАПа.
- **DOSPort** – порт КИП (накопителя ДОС) для считывания реального накопленного значения импульсов ДОС преобразованное в 0.1мм пути (0.1град поворота) пути.

В MRMPrivod.dll включены следующие функции:

Интерфейсные функции MRMPrivod.dll
Procedure PrivodInit(InitMode,Period:word); Инициализирует среду объектов ККО с заданными параметрами. **Параметры:** ***InitMode*** – режим создаваемых объектов ККО 1 – ККО работают по таймеру с периодом ***Period*** мс. Каждый такт считывается и учитывается в накопителе содержимое порта (регистра) ЦАПа **CAPPort**. 2 – ККО работают по таймеру с периодом ***Period*** мс в режиме синхронизации. Каждый такт проверяется содержимое порта (регистра) ЦАПа **CAPPort**. Если это значение отлично от 0 (что означает признак наличия задания на ЦАП), то считывается и учитывается содержимое порта (регистра) ЦАПа. После этого содержимое порта **CAPPort** обнуляется. ***Period*** – периодичность работы таймера ККО, с которой проверяется содержимое ПВВ.
Procedure PrivodCreate(BasePort,CAPPort,DOSPort:word;V_Oborot,Oborot_mm:double); Создаёт новый объект ККО с заданными параметрами и окно его визуализации. **Параметры:** ***BasePort*** – базовый адрес пространства ПВВ. Он же выполняет роль порта синхронизации ***CAPPort*** – адрес порта ЦАП ***DOSPort*** – адрес порта накопителя ДОС ***V_Oborot*** – количество оборотов двигателя за 1 мин соответствующее напряжению на входе ЦАП в 1 вольт ***Oborot_mm*** – количество импульсов ДОС на 1 мм пути (в 1-ом градусе) исполнительного механизма в комплекте двигатель+редуктор

Созданный с помощью PrivodInit() объект ККО эмулирует работу УЧПУ с частотой (тактом) управляющего воздействия 100мс. Таким образом один такт равен 100мс (для реальных систем это слишком медленно! Как правило в реальных системах такт не превышает 1мс.).

В режиме *InitMode=1* ККО выполняет:

С интервалом ***Period*** мс ККО опрашивает порт ***CAPPort***, в соответствии с параметрами V_Oborot, Oborot_mm и значением Ср из порта CAPPort вычисляется пройденный путь St за один такт в 100мс по формуле

St = Ср /1000* V_Oborot /60/10*Oborot_mm

где

Ср/1000 – перевод задания в милливольтах в задание в вольтах;

V_Oborot /60/10 – перевод оборотов двигателя в минуту на 1 вольт в обороты за 1 такт (100мс или 0.1с)

Полученное значение добавляется к накопителю. Значение накопителя переводится в 0.1мм(град) и записывается в порт ***DOSPort***.

Таким образом, если содержимое ***CAPPort*** остаётся отлично от 0 привод (двигатель) постоянно вращается в соответствии с заданным напряжением в ***CAPPort***.

В режиме *InitMode=2* ККО выполняет:

С интервалом ***Period*** мс ККО опрашивает порт ***CAPPort*** как порт синхронизации. Если значение установлено (отлично от 0), то один раз считывается значение Ср содержимого ***CAPPort*** и значение ***CAPPort*** сбрасывается в 0.

Учёт значения из порта CAPPort производится следующим образом:

Значение Ср содержимого ***CAPPort*** рассматривается как заданное за один такт значение перемещения в дискретах ДОС. Значение накопителя увеличивается по формуле:

КИП:=КИП+Ср.

Таким образом накопитель содержит абсолютное перемещение в дискретах ДОС.

Значение накопителя переводится в 0.1мм(град) по формуле:

S = 10*КИП/Oborot_mm

и записывается в порт ***DOSPort***. Таким образом значение Oborot_mm рассматривается как количество дискрет датчика (ДОС) на на 1мм перемещения.

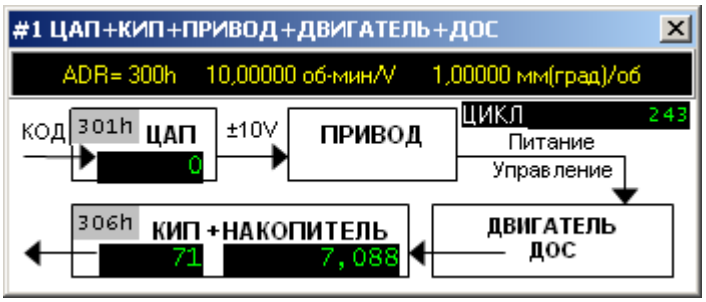
Этот режим позволяет эмулировать синхронизацию формирования и выдачи задания на привод с отработкой задания в приводе каждый такт управления ККО.

Пример:

В ЦАП выдаётся 10 значений

```
for i:=1 to 10  do
begin
 while PortIn($301)<>0 do; // ожидание окончание обработки последнего значения приводом
 PortOut($301,i*100);      // выдача задания в порт ЦАПа
end;
```

Визуальный интерфейс объекта ККО имеет вид



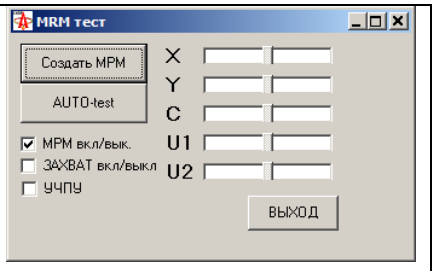
Окно «ЦИКЛ» показывает количество реально отработанных ККО тактов, т.е. таких, при которых прочитанный код из порта ЦАП был отличен от 0.

4.4. Примеры приложений использующих библиотеки МРМ

MRMTestDLL.exe

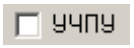
Приложение использует библиотеку MRM.dll.

Создаёт экземпляр объекта МРМ и позволяет управлять (тестировать) координаты МРМ и захват.



Пример вызовов функцмй МРМ:

```
MRMSetC(TrackBar1.Position);
MRMSetU(TrackBar2.Position,TrackBar3.Position);
MRMSetXY(TrackBar4.Position,TrackBar5.Position);
```

Установка флага 

переводит приложение в режим периодического (по прерыванию таймера с частотой 100мс) отслеживания состояния ЦАП-ДОС и входов-выходов для автоматического управления роботом-манипулятором. Процедура обработчика прерывания таймера имеет вид:

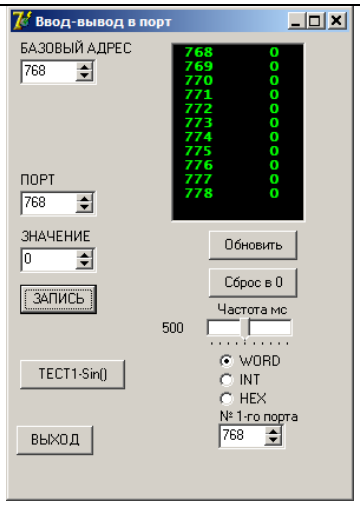
```
var
  TestForm: TTestForm;
  BaseAdr:word=$300;
      IOBaseOutput : word=$300+2;
      IOBaseInput  : word=$300+4;
      IOBaseDAC    : word=$300+6;
      IOBaseDOS    : word=$300+6+5;
      InstrX       : double=0;
      InstrY       : double=0;
      InstrC       : double=0;
      InstrU1      : double=0;
      InstrU2      : double=0;
Var WordP:word;
    IntP :Smallint absolute WordP;

procedure TTestForm.Timer1Timer(Sender: TObject);
begin
  if not StartFlag then exit;
  begin
    WordP:=PortIn(IOBaseDOS+0);
    InstrX:=IntP;
    WordP:=PortIn(IOBaseDOS+1);
    InstrY:=IntP;
    MRMSetXY(InstrX/1000,InstrY/1000);
    WordP:=PortIn(IOBaseDOS+2);
    InstrC:=IntP;
    MRMSetC(InstrC/10);
    WordP:=PortIn(IOBaseDOS+3);
    InstrU1:=IntP;
    WordP:=PortIn(IOBaseDOS+4);
    InstrU2:=IntP;
    MRMSetU(InstrU1/10,InstrU2/10);
    WordP:=PortIn(IOBaseOutput);
    if WordP and $0001 = $0001 then  MRMSetM_On
                               else  MRMSetM_Off;
  end;
```

```
end;
```

MRMIOTest.exe

Приложение использует библиотеку MRM_IO.dll.

Позволяет просматривать содержимое виртуальных портов ввода-вывода в различных форматах (без знаковое и знаковое целое в десятичной и без знаковое в шестнадцатеричной системе счисления) и устанавливать их значения.

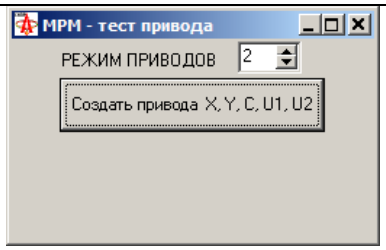


MRMPrivodTest.exe

Приложение использует библиотеку MRMPrivod.dll.

Создаёт пять экземпляров приводов для управления координатами X, Y, C(Uz), U1, U2:

```
PrivodInit(1,100);
PrivodCreate($300,$301,$306,10,1);
PrivodCreate($300,$302,$307,10.5,0.34);
PrivodCreate($300,$303,$308,10.5,1.0);
PrivodCreate($300,$304,$309,10.5,1.0);
PrivodCreate($300,$305,$30A,10.5,1.0);
```



MRMControl.exe

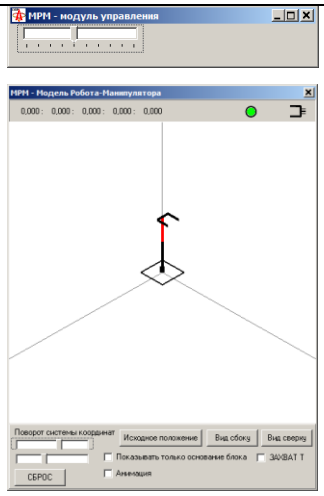
Приложение использует библиотеку MRM_IO.dll и MRM_IO.dll.

Создаёт экземпляр объекта МРМ:

```
IOOpen(BaseAdr);
MRMCreate;
MRMSetOn;
```

По прерыванию таймера считывается содержимого портов ввода-вывода (306h-30Ah) и в соответствии с их содержимым устанавливает значения соответствующих координат МРМ:

```
MRMSetAll( SmallInt(PortIn($308))/10.0,
           SmallInt(PortIn($309))/10.0,
           SmallInt(PortIn($30A))/10.0,
SmallInt(PortIn($306))/10000.0,SmallInt(PortIn($307))/10000.0);
```



5. Основы использование DLL (Delphi, C++, C#)

В этом разделе на примерах показано, как статически (во время запуска приложения) подключаются и используются динамические загружаемые библиотеки (DLL).

5.1. Использование DLL в Delphi

```
// Описание интерфейсов экспортируемых функций
procedure PortOut(PortNo,Value:word);stdcall;external 'MRM_IO.dll';
function PortIn(PortNo:word):word;stdcall;external 'MRM_IO.dll';
Procedure IOOpen(BaseAdr:word);stdcall;external 'MRM_IO.dll';
Procedure IOClose;stdcall;external 'MRM_IO.dll';
procedure PrivodInit(InitMode,Period:word);stdcall;external 'MRMPrivod.dll';
procedure
PrivodCreate(BasePort,CAPPort,DOSPort:word;V_Oborot,Oborot_mm:double);stdcall;external
'MRMPrivod.dll';
……
// вызовы MRM находятся в обработчиках событий нажатия кнопок на форме
procedure TTestForm.Button2Click(Sender: TObject);
begin
  PrivodInit(2,100);
  PrivodCreate($300,$301,$306,1000,600);
  PrivodCreate($300,$302,$307,10,1);
  PrivodCreate($300,$303,$308,10,2);
  PrivodCreate($300,$304,$309,10.5,1.0);
  PrivodCreate($300,$305,$30A,10.5,1.0);
  Button1.Enabled:=False;
  Button2.Enabled:=False;
end;

procedure TTestForm.Button3Click(Sender: TObject);
Var i:integer;
begin
  for i:=1 to SpinEdit1.Value  do
  begin
      while PortIn($303)<>0 do;
      PortOut($301,i*100);
      PortOut($302,1000);
      PortOut($303,500);
  end;
end;
```

5.2. Использование DLL в C++

```
// В пространстве имен System::Runtime::InteropServices хранятся все сервисы,
// отвечающие за взаимодействие с операционный системой.
// В том числе атрибут DllImport, который и позволяет вызывать DLL
using namespace System::Runtime::InteropServices;
// Описание интерфейсов экспортируемых функций
       //Описание функций импортируемых из DLL
       [DllImport("MRM.dll")]
       void MRMCreate();
       [DllImport("MRM.dll")]
       void MRMSetON();
       [DllImport("MRM.dll")]
       void MRMSetAll(double c, double u1, double u2, double x, double y);
       [DllImport("MRM.dll")]
       void MRMSetM_ON();
       [DllImport("MRM.dll")]
       void MRMSetM_OFF();
……
// вызовы MRM находятся в обработчиках событий нажатия кнопок на форме
  private: System::Void button1_Click(System::Object^  sender, System::EventArgs^  e) {
             MRMCreate();
             MRMSetON();
            MRMSetAll(0,0,0,0,0);

                  }
  private: System::Void button2_Click(System::Object^  sender, System::EventArgs^  e)
                 {
                  MRMSetM_ON();
                  }
  private: System::Void button3_Click(System::Object^  sender, System::EventArgs^  e)
                 {
                  MRMSetM_OFF();
                  }
  private: System::Void button4_Click(System::Object^  sender, System::EventArgs^  e)
                 {
                 MRMSetAll(10,20,0,1,1.5);
                 }
```

5.3. Использование DLL в C#

```
// Эта строка подключает сервисы взаимодействия с операционной системой
using System.Runtime.InteropServices;

// Описание функций импортируемых из DLL
   [DllImport("MRM.dll")]
```

```
   public static extern void MRMCreate();
   [DllImport("MRM.dll")]
   public static extern void MRMSetAll(double c, double u1, double u2, double x, double y);

// вызовы MRM находятся в обработчиках событий нажатия кнопок на форме

private void button1_Click(object sender, EventArgs e)
        {
            MRMCreate();
            MRMSetAll (0, 0, -30, 25, -43);
        }

private void numericUpDown1_ValueChanged(object sender, EventArgs e)
        {
            MRMSetAll(0, System.Convert.ToDouble(numericUpDown1.Value), -30, 25, -43);

        }
```

6. Язык управления МРМ

Язык команд МРМ состоит из слов (команд). Каждое слово состоит из одной буквы (адрес слова) и числового значения. Адрес слова сообщает системе МРМ о необходимости того или иного действия. Каждая строка программы содержит код только одной команды. Все перемещения задаются только относительно.

Коды команд управления МРМ приведены в таблице:

Код команды	Назначение
F	Заданная скорость перемещения (в мм/мин) по осям X, Y соответственно
X, Y	Задание на перемещение (в 0.1мм) по осям X, Y соответственно
S	Заданная скорость поворота (в град/мин) по осям Uz, U1, U2
A, B, C	Задание на поворот (в 0.1град) по осям Uz, U1, U2 соответственно
M03, M04	Включить – выключить манипулятор
M02	Конец программы

Пример управляющей программы МРМ:

№ кадра	Команда	Пояснение
N1	F1000	Установить скорость перемещения 1 метр в минуту
N2	S600	Установить скорость поворота/наклона 60 градусов в минуту (один градус в секунду)
N3	X5000	Переместиться по X на 500мм
N4	Y3000	Переместиться по Y на 300мм
N5	B900	Наклонить второй блок на 90 градусов
N6	M03	Включить захват
N7	A1800	Повернуться на 180 градусов
N8	M04	Выключить захват
N9	B-900	Наклонить второй блок на -90 градусов (вернуть в исходное положение)
N10	Y-3000	Переместиться по Y на -300мм (вернуть в исходную позицию)
N11	X-5000	Переместиться по X на -500мм (вернуть в исходную позицию)

N12 M02 Закончить программу

7. Программирование функций управления движением МРМ

В этом разделе будет рассмотрен пример программного кода интерполятора для формирования выдачи задания на привод каждый такт.

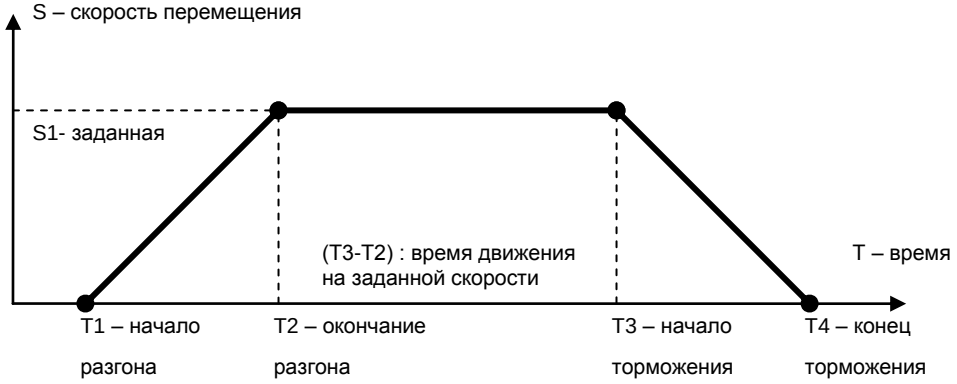
Ставится задача выполнить перемещение по одной из координат на заданное расстояние и с заданной скоростью. Например, отработать кадры УП:

N10 F1000

N20 X5000

При разработке алгоритма учитываются следующие моменты:

1. Расчёты перемещений в УЧПУ ведутся в микронах (мкм)
2. Один такт УЧПУ равен 100мс.
3. В начале отработки кадра перемещения координатная скорость равно 0
4. Если это возможно (т.е. если перемещение достаточно), то необходимо разогнаться по координате до заданной скорости.
5. В конце отработки кадра перемещения координатная скорость должна быть равна 0.
6. Разгон до заданной скорости и торможение до нулевой скорости в конце кадра должны выполняться постепенно с постоянным заданным ускорением за каждый такт (см. рисунок).



Программный код	Назначние
// Блок инициализации	Блок инициализации выполняется один раз перед началом отработки кадра УП.
UZadan:= 5000*100	Формирование значения заданного перемещения из кадра в единицах УЧПУ – перевод из мм в мкм.
ZnakU:=1; if UZadan<0 then ZnakU:=-1;	Определяется знак задания как направление перемещения.
FuZad:= 1000*100/60/10;	Вычисляется заданная скорость в мкм/такт
du := 12;	Задаётся ускорение в мкм/сек/такт (определяется исходя из характеристик привода, двигателя,

	редуктора и т.п.)

TormozU :=False;
iTDU :=0;
iLU :=0;
lTormU :=0;
 	
TaktU :=0;
	Признак начала торможения
	Текущая скорость за такт
	Длина пройденного в кадре пути
	Длина тормозного пути (накапливается как путь разгона до заданной скорости)
	Кол-во тактов разгона до заданной скорости

// Блок расчёта
OstU:=Abs(UZadan-iLU);
 	
if (OstU<=lTormU+Abs(iTDU))
and (not TormozU) then
begin
TormozU:=True;
 	
du:=2.0*(Abs(iTDU)*TaktU-OstU)/TaktU/(TaktU+1);
 	
if du=0 then du:=1;
end;
 	
if not TormozU then
begin
if ABS(iTDU)<FuZad then
begin
if Abs(iTDU+ZnakU*du)>FuZad then
begin
iTDU:=ZnakU*FuZad;
end
else
begin
iTDU:=iTDU+ZnakU*du;
Inc(TaktU);
lTormU:=lTormU+Abs(iTDU);
end;
end;
end
else
begin
if OstU<=du then
iTDU:=ZnakU*OstU
else
iTDU:=iTDU-ZnakU*du;
 	
if abs(iLU+iTDU)>abs(UZadan) then
iTDU:=ZnakU*(abs(UZadan)-abs(iLU));
end;
 	
iLU:=iLU+iTDU;
	Блок расчёта выполняется каждый такт
	Посчитать остаток пути
	
	Проверить нужно ли начинать торможение:
	ЕСЛИ ОТСТАТОК МЕНЬШЕ ЧЕМ ТОРМОЗНОЙ ПУТЬ
	с запасом на один полный такт
	то УСТАНОВИТЬ ФЛАГ ТОРМОЖЕНИЯ
	и РАССЧИТАТЬ СКОРОСТЬ ТОРМОЖЕНИЯ
	из формулы S=V0*t+a*t^2/2
	считая что тормозим со скорости iTDU и
	нужно затормозить за TaktU тактов}
	
	ЕСЛИ НЕ НАДО ТОРМОЗИТЬ
	
	Если не достигли заданной скорости, то продолжаем разгон
	Проверяем, не превысим ли заданную скорость:
	если ДА, то берём заданную скорость
	
	если НЕТ, то
	
	рассчитать новое значение iTDU
	считаем кол-во тактов разгона
	считаем длину пути разгона: будет использоваться как длина тормозного пути
	
	ЕСЛИ НАДО ТОРМОЗИТЬ
	
	Скорее всего это последний такт, значит выдаем все что осталось
	Иначе обычное торможение с рассчитанным ускорением
	
	Проверка на случай если вылетели за конечную точку – корректируем задание
	
	Накопление заданного положения, формируемого УЧПУ

8. Пример использования МРМ для ПО УЧПУ

Распределение адресов портов ввода-вывода

```
IOBase          : word=$300;
IOBaseOutput    : word=$300+2;
IOBaseInput     : word=$300+4;
IOBaseDAC       : word=$300+6;
```

Чтение-запись входов-выходов

```
Procedure Technology;
BEGIN
//  1. ПРОЦЕСС ТЕХНОЛОГИИ   - обработка аварийной ситуации
  if (not TestIns(21,10)) or   //  Нажата кнопка выключения питания
     (not TestIns(17,10)) or   //  Сработал аварийный конечный выключатель
     (not TestIns(19,10)) or   //  Нажали кнопку АВАРИЙНОГО ОТКЛЮЧЕНИЯ 1
```

```
     (not TestInsTest(124,10,1))     // Нажали кнопку АВАРИЙНОГО ОТКЛЮЧЕНИЯ 2
  then
  if _bt.PowerFlag and $1 =$1 then // Если включено питание
  begin
    _bt.PowerFlag :=_bt.PowerFlag and $FFFE; // Сбросить флаг питания
    FillChar(Timers,SizeOf(Timers),0);  //  Обнуление технологических таймеров
    StopAllMove;  // Остановка движения
    for i:=2 to 144 do Setout(i,0);  // Обнуление всех выходов
  end;

//  2. ПРОЦЕСС ТЕХНОЛОГИИ – выполняется всегда
  WriteOuts(1);  // Чтение-запись входов-выходв
  ErrorProc; // Обработчик ошибок
  TEXNProc; // Обработчик технологических функций
END;
```

Процедура чтение-записи входов-выходов

```
Procedure WriteOuts(kak:integer);
// kak=0 - без таймера
//    =1 - с таймером
Var iob1,iob2:byte;
    iow:word absolute iob1;
Var w:word;
begin
// Запись выходов
CommandCriticalSection.Enter;
    iob1:=_BT.RegsOut[0];            // Подготовка из массива регистров выходов
    iob2:=_BT.RegsOut[1];
    PortOut(IOBaseOutput+0,iow);     // Запись
    iob1:=_BT.RegsOut[2];
    iob2:=_BT.RegsOut[3];
    PortOut(IOBaseOutput+1,iow);
// Чтение входов
     iow:=PortIn(IOBaseInput);       // Чтение
    _BT.RegsIn[0]:=iob1;             // запоминаем в массиве регистров входов
    _BT.RegsIn[1]:=iob2;
// если установлен признак синхронизации с устройствами (константа 700),
// то признак того, что можно посчитать и выдать очередное задание
// на перемещение (виртуальный синхро-импульс Sinc) устанавливается в 0
// только если привода прочитали, обработали и сбросили показания портов-ЦАПов
  if (PortIn(IOBaseDAC+0)<>0) or
     (PortIn(IOBaseDAC+1)<>0) or
     (PortIn(IOBaseDAC+2)<>0) or
     (PortIn(IOBaseDAC+3)<>0) or
     (PortIn(IOBaseDAC+4)<>0)
then Sinc:=1
else Sinc:=0;
CommandCriticalSection.Leave;
end;
```

Процедура расчёта и выдачи в ЦАП задания на перемещение

```
Function  Count_PVT:boolean; // считает новое очередное задание в такте
                             // по каждой координате, формирует значение для ЦАП
                             // и записывает их в порт ЦАП
begin
       Result:=True;
       GEOM; // выполнить очередной шаг интерполяции – расчёт TDX, TDY, …, TDU

       CommandCriticalSection.Enter;
       Sinc:=1;       // виртуальный синхро-импульс Sinc=1 – задание записано в ЦАП
       PortOut(IOBaseDAC+0,trunc(tdx));
       PortOut(IOBaseDAC+1,trunc(tdy));
       PortOut(IOBaseDAC+2,trunc(tdz));
       PortOut(IOBaseDAC+3,trunc(tdc));
       PortOut(IOBaseDAC+4,trunc(tdu));
       PortOut(IOBase,1);
       CommandCriticalSection.Leave;
end;
```

Метод Execute потока TCOMCommand. Процедура расчёта и выдачи в ЦАП задания на перемещение

```
// создание потока для интерполяции
  COMCommand  :=TCOMCommand.Create(True);
  COMCommand.FreeOnTerminate:=True;
  COMCommand.Priority:=tpTimeCritical;
  COMCommand.Resume;

procedure TCOMCommand.Execute;
begin
while not Terminated do
begin
// Если УП в отработке – то РАЗБОР КАДРОВ - ОПРЕДЕЛЕНИЕ ЗАДЕЛА КАДРОВ НА ОТРАБОТКУ
   if _BG.UPSost1 and ($1+$2+$4+$8+$10)<> 0 then
   BEGIN
     OPZK(ZadelKadrov);
   END;
   if DO_Count_PVT then  // если установлен флаг «ПОСЧИТАТЬ И ВЫДАТЬ ОЧЕРЕДНОЕ ЗАДАНИЕ»
   begin
     DO_Count_PVT:=False;     // Сброс флага
     Count_PVT;               // Вызов процедуры расчёта и выдачи задания в ЦАП
   end;
   sleep(1); // отпускаем процессор на 1мс
end;
```

Флаг DO_Count_PVT : «ПОСЧИТАТЬ И ВЫДАТЬ ОЧЕРЕДНОЕ ЗАДАНИЕ» устанавливается в CallBack-процедуре обработчика события прерывания мультимедийного таймера (таймера высокой точности в Windows XP или таймера

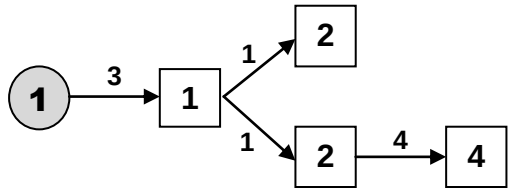
реального времени в ОС жёсткого реального времени) с периодом заданного такта управления (например, 10мс).

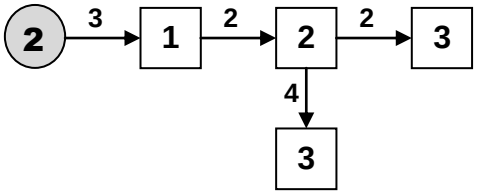
```
Var TESTTimer    : Longword=0;
...
TESTTimer := TimeSetEvent(1, 0, @TimeCallBack1, 0, TIME_PERIODIC);
...
//-----------------------------------------------------
procedure TimeCallBack1(TimerID, Msg: Cardinal; dwUser, dw1, dw2: Longword); pascal;
Var P,V,T,i:longint;
begin
  inc(CNCTime);
  if CNCTime mod 10 = 0 then inc(SysTime); // Системный технологический таймер
  if not DO_Count_PVT_F then
  begin
    if CNST^.wPar.QP[701]<>0  // заданный константой ТАКТ таймера управления в мс
       then DO_Count_PVT_F:=(CNCTime mod CNST^.wPar.QP[701]) = 0
       else DO_Count_PVT_F:=(CNCTime mod 10) = 0;
  end;

 if CNST^.wPar.QP[700]=0 then        // если не установлен признак синхронизации
                                     // с устройствами (константа 700),
 begin
   if DO_Count_PVT_F then
   begin
      DO_Count_PVT:=True;
      DO_Count_PVT_F:=False;
   end;
 end
 else
 begin
   if (DO_Count_PVT_F) and (Sinc =0) then
   begin
      DO_Count_PVT:=True;
      DO_Count_PVT_F:=False;
   end;

 end;
end;
```

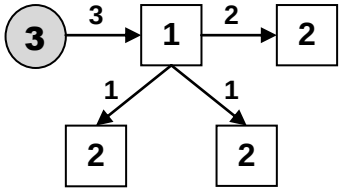
9. Лабораторные работы.

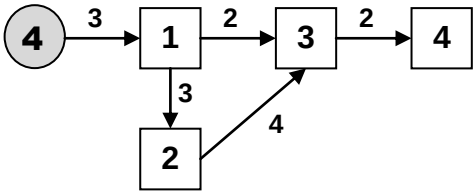
9.1. Лабораторная работа №1: Синхронизация потоков.

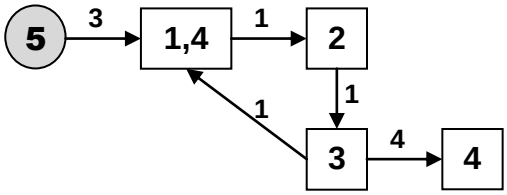
Цель работы: проектирование и разработка многопоточного приложения заданной архитектуры, которая в виде схемы определена в варианте задания:

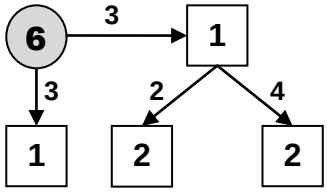


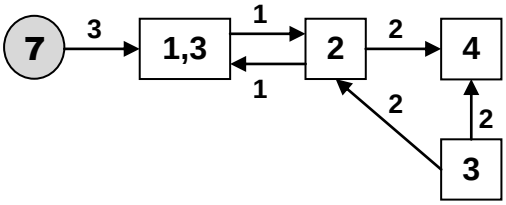


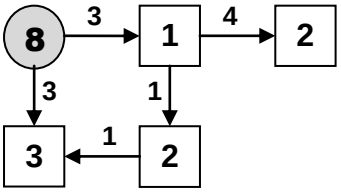


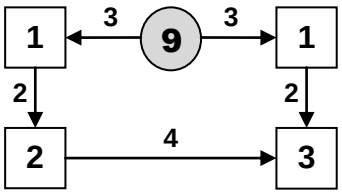


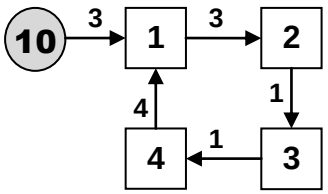




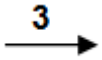








На схеме приняты слежующие обозначения:

Символ	Описание
	Основной (главный) поток процесса. Он предназначен для создания всех дочерних потоков процесса, объектов синхронизации и т.п. Этот поток и даёт старт работы синхронизированной многопоточной системе. Число в круге обозначает номер варианта.
	Дочерний поток процесса. Число в квадрате обозначает порядок активации потока во времени при старте и функционировании системы.
	Стрелка указывает на синхронизированную связь двух потоков. Направление определяет отношение «главный» → «подчинённый». Цифра определяет способ синхронизации: 1 – критическая секция 2 – мьютекс 3 – событие 4 – семафор

Действия, выполняемые главным и дочерними потоками, необходимо придумать самосточтельно. Это могут быть, нгапример, вычисления, графические отображения, чтение-запись данных на диск и т.п. Основное требование – главгый поток должен выводить на экран необходимую информацию, подтверждающую корректность работы приложения.

По результатам работы должен быть представлен отчёт, содержащий постановку задачи, описание выполненной работы и программный код системы. На защиту работы представляется функционирующие приложение.

9.2. Лабораторная работа №2: Разработка модельного ПО УЧПУ для МРМ

Цель работы заключается в проектировании и разработке ПО с архитектурой и функциональностью которая моделирует некоторые функции реального УЧПУ. Объектом управления является МРМ.

ПО УЧПУ должно включать 3 потока, описание назначения которых приведены ниже (названия потоков выбраны условно).

КОНСОЛЬ (CONS) – поток обеспечивает интерфейс взаимодействия с оператором. Должен выполнять следующие задачи:

1. Обеспечивать интерфейс с пользователем в режиме ожидания возникновения управляющего события: нажатие кнопки, ввода данных и т.п.
2. Ввод УП как из файла, так и в режиме редактора
3. Разбор УП – перевод текста во внутреннее представление, пригодное для отработки. В результате должен быть сформирован буфер отрабатываемой УП (БУП), где каждый элемент (объект, запись …) содержит всю необходимую информацию для отработки одного кадра УП.
4. Выдача команды на отработку, остановку (приостановку, продолжение) УП.
5. Формирование ручного управления всеми координатами и устройствами МРМ.

ГЕОМЕТРИЯ (GEOM) – поток работает по прерыванию таймера (функции таймера) с периодичностью управляющего воздействия на МРМ и обеспечивает:

1. Каждый такт проверяет наличие команды от процесса CONS на отработку, прерывание (приостановку, продолжение) УП и отрабатывать команду.
2. При возникновении команды на отработку УП устанавливает на выполнение УП из БУП и начинает формировать каждый такт задание на перемещение для текущего кадра или установку на выполнение технологической программы.
3. УП из БУП выполняется кадр за кадром. Переход на выполнение следующего кадра выполняется только после полного выполнения предыдущего кадра. Перемещение должно выполняться с разгоном и торможением в соответствии с принципами приведёнными в п.5.

ТЕХНОЛОГИЯ (TECHN) – поток обеспечивает выполнение технологических функций:

1. Проверяет наличие внешних сигналов через устройства дискретных входов (портов ввода-вывода). Таким сигналом для МРМ может быть сигнал аварии (устанавливается в 1 бит аварии на одном из портов). При возникновении аварии необходимо установить команду на остановку отработки УП (она должна быть подхвачена и отработано процессом GEOM) и прекратить выполнение всех технологических функций.
2. После установки на отработку технологических функций M03, M04 (установленных потоком GEOM) выполняет их обеспечивая при этом параллельную отработку перемещений и техфункий.

По результатам работы должен быть представлен отчёт, содержащий постановку задачи, описание выполненной работы и программный код системы. На защиту работы представляется функционирующие приложение.

Литература

1. Введение в многопоточность. http://www.hardline.ru/1/5/1530/
2. Рихтер Дж. Windows для профессионалов: создание эффективных Win32 приложений с учетом специфики 64-разрядной версии Windows / Пер, англ - 4-е изд. - СПб; Питер; М.: Издательско-торговый дом "Русская Редакция", 2001. - 752 с.; ил.
3. QNX Neutrino Operating System. http://www.qnx.com/developers/docs/6.3.2/neutrino/
4. В. Денисенко и др. Испытания электронной аппаратуры: быстро и эффективно. // Компоненты и технологии. – 2004. - №4 . http://www.kit-e.ru/articles/device/2004_4_216.php
5. ПРОГРАММНОЕ ОБЕСПЕЧЕНИЕ EBI (HONEYWELL). http://www.concept-pro.ru/cp/-ebi-honeywell/46-ebi
6. Предложение для автоматизации машин и механизмов на базе инновационной архитектуры MachineStruxure. http://www.is-com.ru/machine_struxure.html
7. Автоматизированная система управления климатом в тепличных хозяйствах. http://zaz.gendocs.ru/docs/2800/index-570697.html?page=5
8. Система числового программного управления CNC4000. http://www.zont.com.ua/production/cnc4000
9. Энциклопедия АСУ ТП. http://bookasutp.ru/Chapter5_2.aspx
10. Крапивный Ю.Н., Крапивная О.В. Программное моделирование робототехнических систем. Тезисы доклада Всеукраинского научно-методического семинара "Информационные технологии в учебном процессе". Одесса, 2009г., С. 14-18.
11. Работа с потоками (C# и Visual Basic). MSDN. http://msdn.microsoft.com/ru-ru/library/ms173178