Legend:

Grey – consider revising

**Core** – the very fundamental part of the whole system. It consists of its kernel and its modules. Here’s what the core *does*: it lets us program any system (the system that lies outside of the “Core”) and control it.

We provide a possibility for a user to control some parts of the core with **configuration files (henceforth referred as “configs”)**.

**Important notice**

For addressing, we use IDs. But to make it look more convenient for a client, we build a “bidirectional dictionary” to match IDs and their aliases. **And we use domain-addressing system**.

**Control modes**

There are different modes of controlling (by now, we’ve only been implementing the first mode):

1. Manual – in this mode, a real user operates all the controlling.
2. Semi-Automatic – in this mode, some of the controlling tasks are operated automatically.
3. Automatic – I bet you guess what it does :D

**Components**

Every component has to have its own **unique** ID.

1. Kernel – the main supervisor and the main transport node.
   1. Supervisor – initializes and reinitializes all the core modules. It also can, using specific commands, tell any core modules to reboot the very non-core modules they dispatch.
   2. Transport node – sustains transportation of messages between core modules.
2. Core modules – the core functional lies exactly in these modules.
   1. Client Manager (CM) – authorizes those users, who want to connect, processes their input and informs them (via GUI, or console, or whatever…) about the current system state. Also, this is where we match names or aliases with their corresponding IDs.
   2. System Control Manager (SCM) – sustains all the control stuff. This is like a CPU for the connected system. One of the most important things it does is that it translates *commands* into *instructions*. **Instructions** – the atomic (meaning indivisible) control units that are invoked by the very system the “Core” is connected to. **Commands** – the control units of higher abstraction level, comparing to instructions. Commands are used to program the system and to control it in runtime (as an example, via console).
   3. External Input Manager (EIM) – receives any data (from the system we communicate with – it is “communicate”, not “control” because we may have no rights to control it) that needs to be received. I say “any data”, because absolutely nothing else receives this data. And it **does not send anything** to the system.
   4. Remote System Dispatcher (RSD) – sends control commands to the system. If we cannot control the system (for any reason) nobody is gonna know that nothing had happened when we sent off a command, because RSD **does not receive any data**.

How data transportation between modules work:

**Communication**

Communication within Core is sustained by Kernel – every piece of data, that moves from one module to another, or to some storage, or wherever – it goes through the Kernel. If a piece of data moves from Kernel to some module, we use IKernelCommunicator (see the description below) to handle it.

**Domain addressing system**

First, we take ITransferable to the Kernel. After that, we check if the first element ICollection<object>, caster to UInt32 is zero. If it is zero, we’ve reached our destination. If it is not, we lookup this value in the current addressing table (current means that it is located in the current node) and if there is such ID, we first remove this element from the collection, and then we pass it one level down the addressing tree – to the IAddressable, matched to the destination ID.

**External Input/Output**

We use DLLs (some libraries) to communicate with external IO interfaces. In those DLLs there must be ReceiveFrom( whence, bytearray ) and SendTo( whither, bytearray ) functions.

Since we don’t know what devices and external IO interfaces we are going to have on the other side, we just store in our “Device” objects a pointer (64-bit unsigned integer) to the memory, where the descriptor for a particular device is located – those are the “whence” and “whither” parameters – they have ULONGLONG type. And, obviously, bytearray is the very data we send/receive.

First, how we receive data from the system, connected to Core:

First, we call IIncomingDataSource.Listen() method in a separate thread. It works

**Interfaces**

Our philosophy is that all parts of the Core are completely replaceable. To replace the part you want, you just need to implement a certain interface.

Interfaces are stored in a single directory, each interface in its own file.

**Zero-generation:**

* enum TransferableDataType – used in ITransferable to determine, what data we transfer
* IAddressable – all core components must have this
  + uint **ID**{get;set;} – as mentioned above, addressing in the Core uses ID numbers
* ITransferable – can transfer data within Core
  + UInt32 DestinationID{get;set;} – ID of the recipient
  + ICollection<object> Data{get;set;} – contains the very data we want to transfer
  + TransferableDataType DataType{get;} – see the description of TransferableDataType
* IKernelCommunicator – used to sustain communication between Kernel and Modules
  + delegate void SendToKernel(ITransferable) – called when IKernelCommunicator receives some data from remote source to take the data to Kernel

**First-generation interfaces**

* ICommand – used to transfer commands between modules:
  + uint DestinationID{get;set;} – ID of the guy, who’s gonna receive this
  + ICollection<object> Command{get;set;} – we don’t know, how the recipient is gonna sort out and invoke those commands, so we just say that it is some collection of some objects
* IInternalEvent – describes an event, using an event ID from a local base
  + uint SourceID{get;set;} – ID of the guy, why raises the event (his ICoreComponent unique ID)
  + uint EventID{get;set;} – an ID from special base
  + byte[] AdditionalInfo{get;set;} – optional additional information about the event
* IIncomingData : ITransferable – a container for some data, that “arrives” - stores its source ID and the data itself
  + UInt32 SourceID – the ID of the guy, who has sent us this block of data
  + *+Inherited members*
* IInstructionPacket : ITransferable – used to transfer instructions from ISystemControlDispatcher to IRemoteSystemDispatcher and to send these instructions to the destination (device) to be invoked
  + *+Inherited members*

**First-generation interfaces:**

* IControllable – can invoke ICommands
* IController – dispathches ICommands (to a specified target) to be invoked
* IClient – manages a connected client – contains information about his access level and supports the very data flow between Core and him
* IDevice – can invoke instructions
* IPinger – the one who pings IPingable*s*
* IPingable – can be pinged (most commonly, by kernel – it pings all modules and storages)
* IIncomingDataSource – a listener (receiver) of incoming messages. Like Socket, can receive and close. Not send.
* IEventsBaseUser – can store IInternalEvent*s* in IEventsBase. **Has a link to the base it uses**
* IEventsBase – a storage for all IInternalEvents
* IIncomingDataStorage – a storage for the data, that comes from outside, through IExternalInputManager
* IIncomingDataStorageOwner – the one that has a link to IIncomingDataStorage. So, this guy can to whatever he wants with it.
* IIncomingDataStorageReader – can read from IIncomingDataStorage – we don’t know yet, how they’re gonna do this – sequential access, random access,..
* ILoggable – can log some data. Uses an ILogger provided