OOWG Tutorial

Design of base ICS classes

T. Herlin

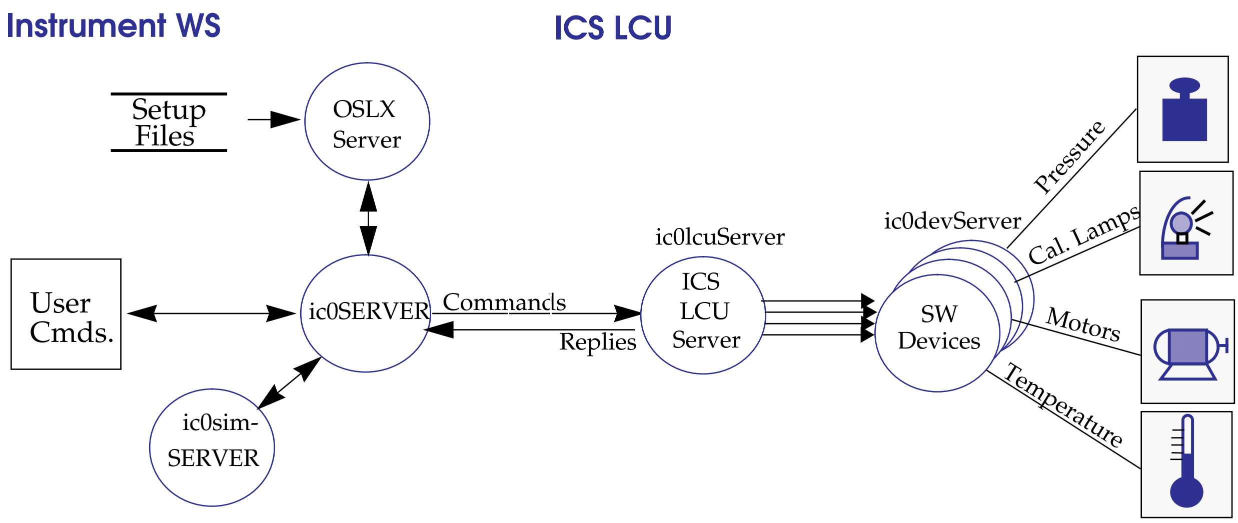
28/05/96

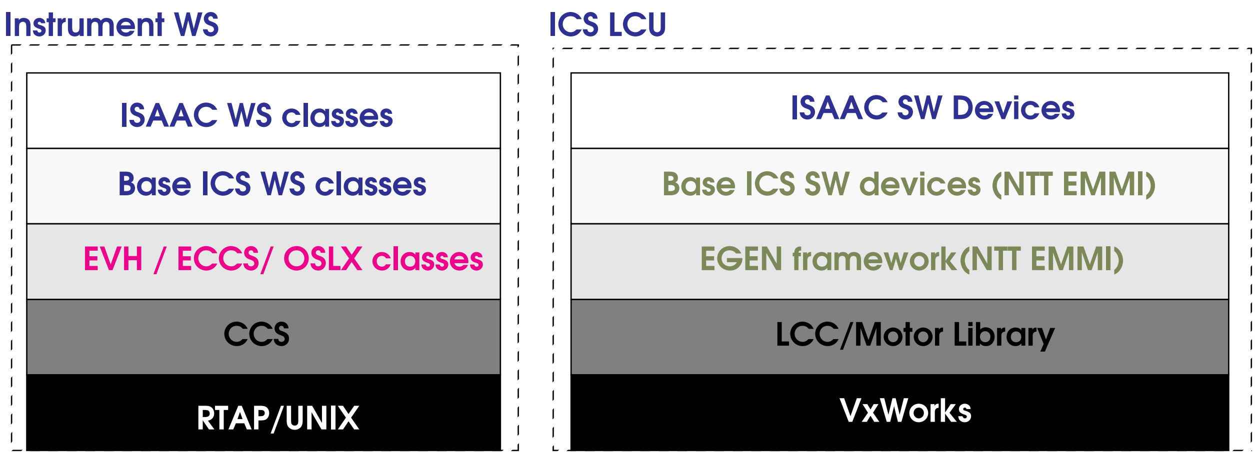
Contents

Objective

- To establish a common set of classes used by instrument control SW as a platform for building new systems.
- Conceived by **ISAAC ICS** and **EMMI** upgrade project.
- "NO-COST" spin-off
- The classes should be viewed as "Guidelines" not as "the Rule"
- Open classes - input from other instruments wanted (and necessary)

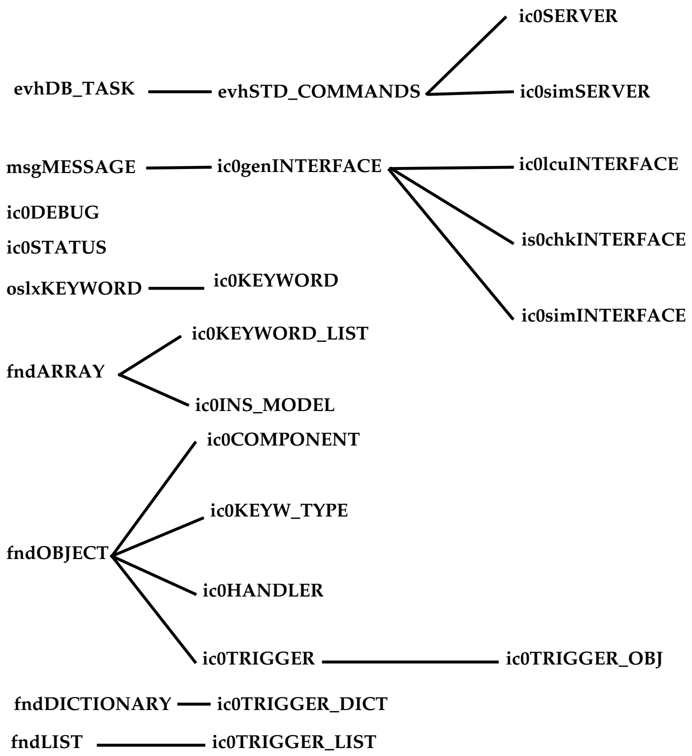
ICS Architecture





IC0 WS Class Overview

IC0 WS classes derived from: evh, fnd and oslx



- **ic0SERVER** - base for ICS WS Server process
- **ic0simSERVER** - base for ICS Simulator process
- **ic0genINTERFACE** - generic interface to other processes

WS Application Framework

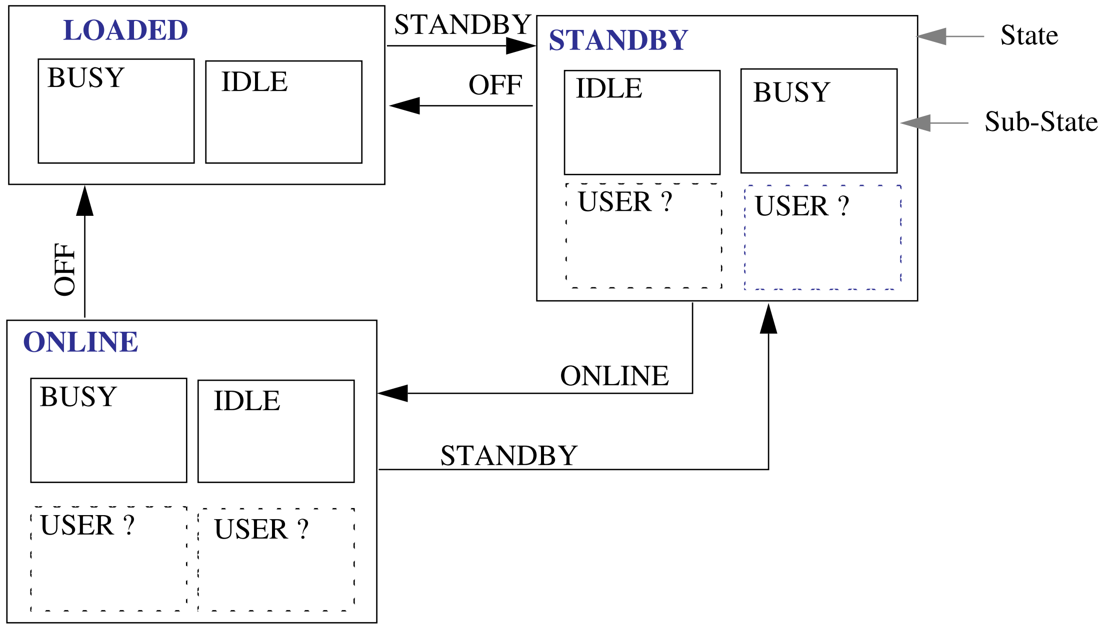
Why Framework ?

- Basic set of commands supported
- Commands can easily be added
  – via sub-classing of ic0SERVER
- Command behaviour can be modified
  – handling functions can be overloaded
  – “dummy” functions inserted

Supported ICS standard commands:

- **State** commands:
  – OFF (PDOWN), STANDBY, ONLINE (OPERATING)
- **Control** commands:
  – SETUP, STATUS, INIT, STOP, DISABLE, ENABLE, LOCK, UNLOCK, SETMODE.
- **Q**: Why not more commands ?
  – **A**: INS Common SW Spec !

ICS STATES



- **CheckSubstate()** - function to determine level of parallelism
  - Normal commands prevented
  - Emergency commands allowed
  - Alarms and DbEvents handled
- **ResetSubState()** - resumes IDLE substate.
- Sub-class overload adapts the behavior !
- Transition functions supported e.g. Loaded2Standby()

Using cmd to parse parameters

- CCS/cmd provides functions to parse command body parameters.
- Supported by: **cmdParam***XXX* functions
- CDT information used for command parsing.

- **cmdParamList** - parse parameters into cmdPARAM_LIST structure.
  - each parameter is tagged by type and if multiple parameters the number and array of values is attached.

- **cmdParamGetBy**(Name | Index) retrieves values from parameter list converted to the type.
  - Note: *Memory allocated and re-used internally*

- **cmdParamSetBy**(Name | Index) changes values from type to parameter list values.

Using cmd to parse parameters (cont'd)

- **cmdParamDefault** build up param. list from CDT with default parameters.
  – Useful for systems sending commands.

- **cmdParamDelete** delete a parameter from param. list.
  – Useful for forwarding commands.

- **cmdParamToMsg** build up message body from param.

Command Example

Adding CONFIG command to IC0

```
#include "ic0SERVER.h"
// Example of a simple sub-class of ic0SERVER
// A evh callback for the command CONFIG is added.
class testSERVER : public ic0SERVER
{
public:
    testSERVER(const dbSYMADDRESS dbRef,
                     ccsPROCNAME   chkServerName, ccsPROCNAME   simServerName,
                     vltINT32      lcuNum, ic0lcuSERVER *lcuServers,
                     char          *dictionary, char          *alias);

    evhCB_COMPL_STAT ConfigCB(msgMESSAGE &msg, void *udata);
    evhCB_COMPL_STAT ConfigReplyCB(msgMESSAGE &msg, void *udata);
};

testSERVER::testSERVER(const dbSYMADDRESS dbRef,
                     ccsPROCNAME   chkServerName, ccsPROCNAME   simServerName,
                      vltINT32      lcuNum, ic0lcuSERVER *lcuServers,
                      char          *dictionary, char          *alias)  :
    ic0SERVER(dbRef, chkServerName, simServerName,
              lcuNum, lcuServers, dictionary, alias)
{
    evhMSG_TYPE_KEY key(msgTYPE_COMMAND);
    evhOBJ_CALLBACK cb(this);

    // isiServer commands
    evhHandler->AddCallback(key.Command("CONFIG"),
                            cb.Proc((evhCB_METHOD)ConfigCB));
}
```

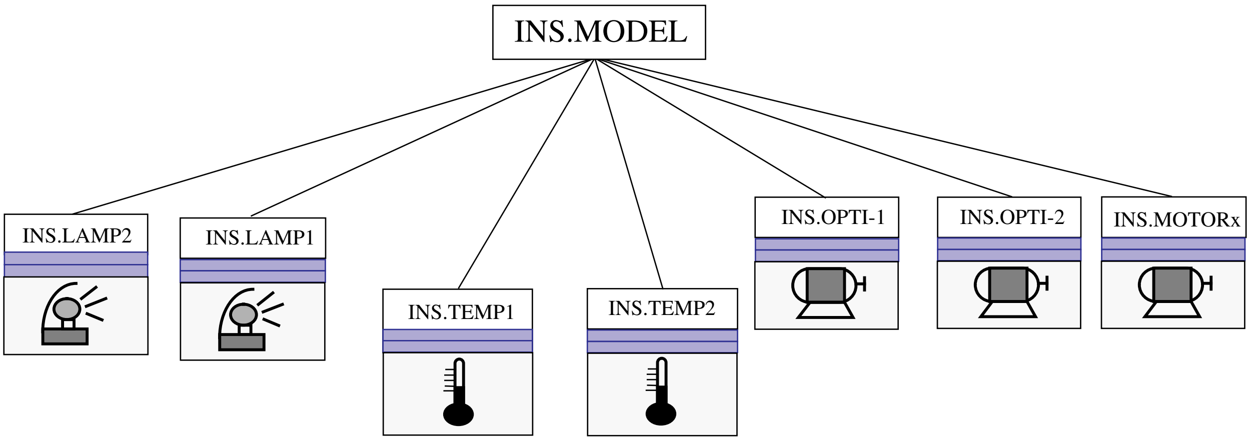
Command Example (cont'd)

```
evhCB_COMPL_STAT testSERVER::ConfigCB(msgMESSAGE &msg, void *)
{
    int                 pNo;     // number of parameters
    char                **sPtr; // pointer to string paramter array
    ic0KEYWORD_LIST     *keywordList;  // list of keywords handled by command

    // Check valid system state.
    // This command is only accepted when system is not in ONLINE
    if (State() == ic0STATE_ONLINE) {
        ErrAdd(ic0MOD,ic0ERR_INVALID_STATE, __FILE_LINE__,
               ic0STATE_LOADED_OR_STANDBY_STR, StateName());
        return ErrorReply(msg,ccsFALSE);
        }

    // Check the substate of the system.
    // Here we determine if the systems sub-state allows to handle this
    // command.
    if (CheckSubState(msg.Command()) != SUCCESS)
        return ErrorReply(msg,ccsFALSE);

    // Parse command parameters into a parameter list structure.
    // When command parameters are not OK return with failure
    if (cmdParamList(msg.Command(), msg.Buffer(),strlen(msg.Buffer()),
                     &paramList, &ErrStack()) == FAILURE) {
        ErrAdd(ic0MOD,ic0ERR_PARSE_PARAMETERS,
                __FILE_LINE__,msg.Command());
        return ErrorReply(msg);
        }


    // Handle "load" parameter and extract configuration filename
    cmdParamGetStringByName(&paramList, "load",
        &pNo,(char **)&sPtr, &ErrStack());
    if (pNo == 1) {
        // Processes configuration file sPtr[0]
```

```
        return OkReply(msg);
        }
    // Handle individual function configuration
    cmdParamGetStringByName(&paramList, "function", &pNo,(char **)&sPtr,
                            &ErrStack());

    if (pNo != 0)
        {
        chkServer->ClearBuffer(ic0chkCHECK_OSLXBUF);
        // check the keywords
        if (CheckFunctions("CONFIG", pNo, sPtr, &keywordList,
                           ic0chkCHECK_OSLXBUF) == FAILURE) {
            // The function must be invalid - reply with the error
            return ErrorReply(msg);
            }
        // Save the original received message in forward object
        ic0genFORWARD *forwardObj = new ic0genFORWARD(msg);

        //  Install callbacks for: reply, errorReply and callback
        ccsCOMPL_STAT stat;
        stat = lcuCmd->Forward("CONFIG", &paramList, keywordList,
                               forwardObj, (evhCB_METHOD)ConfigReplyCB,
                               (evhCB_METHOD)ErrorCB,        // use default
                               (evhCB_METHOD)TimeoutCB);     // use default
        // done with the keywordList
        delete keywordList;
        if (stat != SUCCESS)
            return ErrorReply(msg);

        // The reply is first sent when the sub-system replies back
        return evhCB_NO_DELETE;
        }

    // If parameters where not catched yet return with error
    ErrAdd(ic0MOD,ic0ERR_PARSE_PARAMETERS, __FILE_LINE__,"no specified");
    return ErrorReply(msg);
}
```

ICS Instrument Model

- Describing the physical instrument functions as software components.
- Inserting the components into a model - a set of components.
- Short FITS "Keyword's" are used to adress the components in the model.



Advantages

- Organizing of database and it's access
- Dynamical check of Keywords
- Handling instrument modes
- Modifying function keywords i.e. automatic modes

An Instrument Component

Basically a C++ describing the component:

- An active device - e.g. motor

or

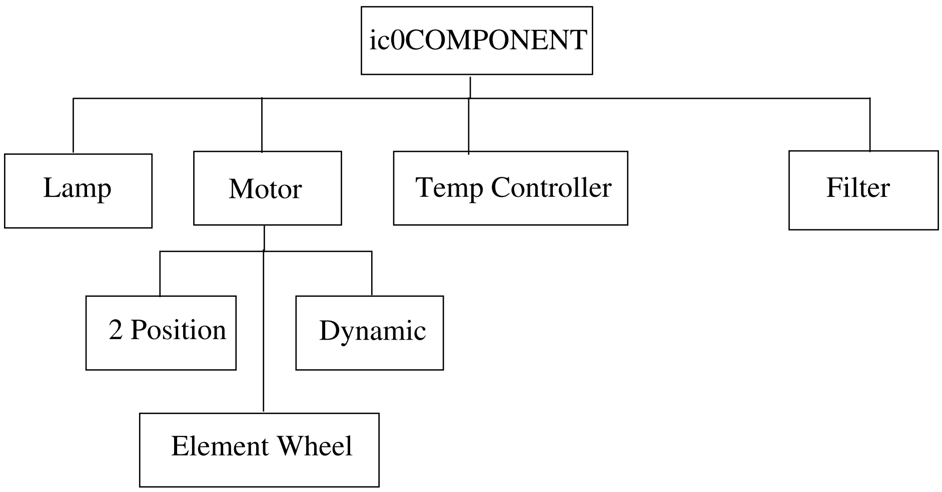
- Passive component - filter wheel.

- Must have a **unique** Keyword.

- Holds database reference of device

- Class derived from **ic0COMPONENT**

– Functional inheritance provided via C++



Keyword Representation

- **ic0KEYWORD** is the class representing a short FITS keyword.

- sub-class of oslxKEYWORD but add's
  - fndOBJECT capabities
  - LCU destination (used for handling systems with multiple LCU's)

- **ic0KEYWORD_LIST** dynamical array of ic0KEYWORD's.
  - array features (fndARRAY)
  - new data can be placed anywhere

- when data is passed between ic0 object's it's either ic0KEYWORD's or ic0KEYWORD_LIST's

Keyword Triggers

- **Callback function** called when a certain keyword is requested e.g. SETUP cmd.
- Mechanism similar to evh command handling.
- Keyword Triggers are registered into a dictionary in **ic0HANDLER**.
- Keyword list's (ic0KEYWORD_LIST) are dispatched by ic0HANDLER
- A ic0HANDLER is member object of the instrument model **ic0INS_MODEL**
- Keyword Triggers can be assigned to any command or a specific command

Model Example

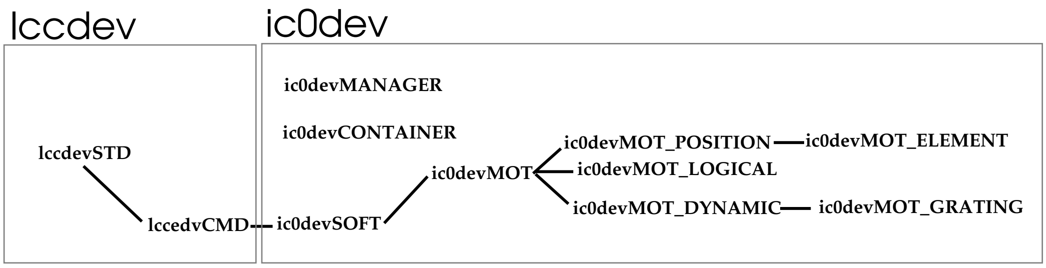
```
class testMOTOR: public ic0COMPONENT
{
  public:
    testMOTOR(ic0HANDLER *handler, char *keyw, char *cmd = "");
    ccsCOMPL_STAT MotorCB(const ic0KEYWORD &keyword, void*);
    ccsCOMPL_STAT GetKeywords(vltINT32 *no, ic0KEYWORD **keywPtr,
            msgCMD cmd);
  protected:
    ic0KEYWORD motorKeyw;
}; /* end class testMOTOR */

class testINS_MODEL: public ic0INS_MODEL
{
  public:
    testINS_MODEL();
    ccsCOMPL_STAT InsModeCB(const ic0KEYWORD &keyword, void*);
    ccsCOMPL_STAT NewKeywordList(ic0KEYWORD_LIST *list, msgCMD cmd);
    ccsCOMPL_STAT GetKeywords(vltINT32 *no, ic0KEYWORD **keywPtr,
            msgCMD cmd);
private:
    ic0DEBUG   debug;
    ic0KEYWORD modeKeyw;
};

testINS_MODEL::testINS_MODEL() :
    modeKeyw("INS.MODE   UNKNOWN")
{
    ic0KEYW_KEY key("");
    ic0TRIGGER_OBJ   trigger(this);

    // Add a INS.MODE keyword trigger
    handler->AddTrigger(key.Keyword("INS.MODE"),
                        trigger.Proc((ic0TRIGGER_PROC)InsModeCB));
    // Add some motor components to the instrument model
    Add(*(new testMOTOR(handler,"INS.OPTI-1")));
```

```
    Add(*(new testMOTOR(handler,"INS.OPTI-2")));
    Add(*(new testMOTOR(handler,"INS.OPTI-3")));
}

testMOTOR::testMOTOR(ic0HANDLER *handler, char *keyw, char *cmd)
{
    char newKeyword[32];
    Id(keyw); // Store the keyword in the device - used as reference
    strcpy(deviceType,"MOTOR"); // store the device type
    ic0KEYW_KEY key("");
    ic0TRIGGER_OBJ   trigger(this);

    handler->AddTrigger(key.Keyword(keyw).Command(cmd),// Add INS.OPTI-i
                        trigger.Proc((ic0TRIGGER_PROC)MotorCB));
    sprintf(newKeyword,"%s.NO", keyw);// reg. INS.OPTI-i.NO keyword trigger
    handler->AddTrigger(key.Keyword(newKeyword).Command(cmd),
                        trigger.Proc((ic0TRIGGER_PROC)MotorCB));

    sprintf(newKeyword,"%s.ID", keyw);// reg. INS.OPTI-i.ID keyword trigger
    handler->AddTrigger(key.Keyword(newKeyword).Command(cmd),
                        trigger.Proc((ic0TRIGGER_PROC)MotorCB));
}

ccsCOMPL_STAT testMOTOR::MotorCB(const ic0KEYWORD &keyword, void*)
{
    Valid(ccsTRUE);// Device function is valid - flag to indicate for the model

    motorKeyw = keyword;// assign the keyword to the device keyword
    return SUCCESS;
}
```

ICS Classes on LCU

Classes witout C++ - how ?

LCU Class Hiearchy



- Egen - code generation framework
– developed by **Ricardo Schmutzer**

- Language describing services i.e commands in a uniform .gen file
- Code generation utility
– Produces: .c, .h, .cdt, .cit, .dbl, .boot
- ***inherit*** construct provides simple inheritance
– table driven inheritance (virtual pointers)

- **auto** and **lcctoo** support modules

Egen - example

```
Library myIcs {
    Module myDev
}
Module myDev {
    Description {
        Simple device build with egen
    }
    Environment myENV     // environments in this module
    Process myPROC        // processes in this module
    Class myMANAGER       // my device manager
    Class mySW_DEVICES    //
    Class myDEVICE        // software device
}

Environment myENV {
Process myPROC myServer
}
Process myPROC {
Class myMANAGER     myManager
}
Class myMANAGER {
    Id 201
    Interface {
        inherit ic0devMANAGER
        Exception Dummy {
            Type Severe
            Format "Dummy error"
        }
    }
    Implementation {
        CodeType CI

        Attribute mySW_DEVICES  swDevices         // all SW devices
    }
}
```

```
Class mySW_DEVICES {
    Id 203
    Implementation {
        inherit ic0devCONTAINER
        CodeType C
        Attribute myDEV         dev1
        Attribute myDEV         dev2
        Attribute myDEV         dev3
    }
}
Class myDEVICE {
    Id 310
    Interface {
        inherit ic0devSOFT

        Exception ReadOldb {
            Type Severe
            Format "Device %s: Unable to read %s from OLDB."
        }
    }
    Configuration { // Database decription of device
        Attribute string<16>    Id
        Attribute int32         myNum
        Attribute Table         myTable[1] {
            string<256> myName
            string<32>  mydevice
            uint32      value
        }
    }
    Implementation {
        CodeType C
        // Overloaded LCC Software Device user functions
        Service UsrInit                 Inherit
        Service UsrSelfTest             Inherit
        Service UsrTest                 Inherit
        Service Setup                   Inherit
    }
} // _o0o_
```

LCU classes

- **ic0devMANAGER**
  - implements device manager
  - SW devices registered as objects
  - SW device commands called as tasks

- **lccdev**
  - implements standard commands
  - basic SW device database e.g. state placeholders for user functions

- **ic0devSOFT**
  - implements INS commands
  - command parameter parsing, description in local DB

ic0devMOT class

- motor specific commands and handling of MCM

- database definition with all information for the motor

- motor device = single or pair of motors

- action table - allowing a sequence of actions
  - motMove parameters changeable via action table

- position table - usefull for driving element wheels e.g. filterwheels.

- SETUP parameters parsed via ic0devSOFT

Conclusion

- Application Framework for ICS development
- ICS Model support - assist in organizing data and handling keywords.
- ICS Classes on LCU possible

IMPROVEMENTS

- Needs to be applied for other instruments.
- Use of standard C++ libraries
- Performance (handling of setup files internally)
- Support for special LCU environment ??
- C++ for LCU should be considered.