A Reusable Driving Engine for Complex Control Systems Based on Embedded TCP/IP

Zhichun Zhang, Jingxin Xiao, Wen Xu, Zhenpen Zhao, Shengquan Tang, Jian Bu
Military Simulation Institute, Aviation University of Air Force, Changchun, 130022, China

Abstract—**This paper presents a complex control system in which a large-scale of devices are controlled by micro control units (MCUs) based on embedded TCP/IP, with emphasis on the management of these MCUs through one or more reusable driving engines. The number of the driving engines bases on the scale of the devices and the real-time requirement. For most cases, one driving engine with one or more NICs （Network Interface Cards） is enough. Each device is controlled by a MCU to sample the device and the sampled data is delivered to applications through the driving engine that receives the driving data from the applications and transmits it to the MCU to drive device. The driving engine running on an industrial personal computer(IPC) communicates with the MCUs through the lower network, a TCP/IP network, while it communicates with the applications through the upper network, a TCP/IP network and(or) a reflective memory (RFM) network. The driving engine is a reusable software architecture and any number of MCUs can be integrated seamlessly in DLL(Dynamic Link Library) without any need to modify the architecture. To develop a new device control system, the only tasks is to implement the MCU embedded routines that must be done whenever any development methods applied and the MCU DLLs and then integrate the DLLs with the architecture. This MCU-based control method fits the man-in-loop simulators best with the most advantage of making the system wiring simple, elegant and clean than AD cards usually used and has some other beneficial features of software reusability, valid precision maintenance, more developing efficient, run-time and reliable performance.**

Keywords-micro control unit (MCU); Embedded TCP/IP; Driving Engine; software reusability.

I. Introduction

In many man-in-loop simulators, such as flight simulators, ship simulators, power station simulators, the trainee operates on the devices and the devices echo. For a system with a small number of devices, Analogue/Digital (Digital/Analogue) cards can perform better for controlling the devices, but for a system with a large number of devices, the system wiring would be most dense and mass, which is not conducive to the system development and maintenance. Besides, in engineering practices with controlling devices, the software usability and the developing mode are not be paid attentions and these lead to many problems, such as repeated development of hardware controlling software, mutual interdependence between hardware developers and software developers, device precision maintenance invalid due to device wearing. Now the technology of embedded TCP/IP or integrating TCP/IP stacks into embedded system insures the connection between embedded systems and Ethernet based on TCP/IP. This technology is used widely in industry control fields but its application in simulators is not reported[1][2][3]. This paper presents a complex control system in which a huge mass of devices controlled by micro controller units (MCUs) based on embedded TCP/IP, with emphasis on the management of those MCUs through one or more reusable driving engines. This device control method has the features of reusable driving architecture, more developing efficient, easily maintenance, run-time and reliable performance and fits the man-in-loop simulators best with the most advantage of making the system wiring simple, elegant and clean than AD cards usually used.

© 2015. The authors - Published by Atlantis Press

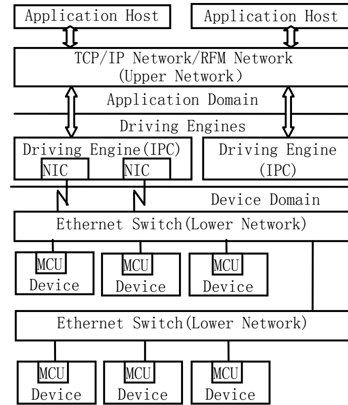

Figure 1. Hardware structure

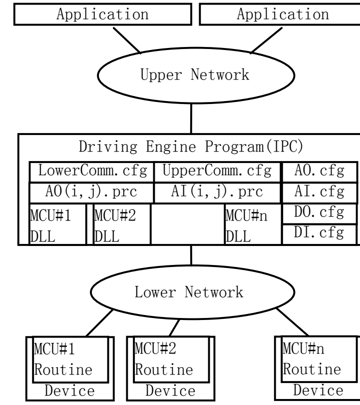

Figure 2. Software structure

II. OVERALL STRUCTURE

Figure.1 shows the overall structure of the device control system based on embedded TCP/IP. There are three types of partners, the MCUs, the driving engines and the applications. The number of the driving engines bases on the mass of the devices and the real-time ability. For most cases, one driving engine with one or more NICs（Network Interface Cards）is enough. Each device is controlled by a MCU to sample the device and the sampled data is delivered to applications through the driving engine that receives the driving data from the applications and transmits it to the MCU to drive device. The driving engine running on an industrial personal computer(IPC) communicates with the MCUs through the lower network, a TCP/IP network[4][5], while it communicates with the applications through the upper network, a TCP/IP network and(or) a reflective memory (RFM) network[6]. The driving engine is the proxy of all MCUs and the applications communicate only with the driving engine but not any MCUs. Usually the device domain, such as aircraft cabin, is far off the applications and(or) the driving engines. Fortunately, the embedded TCP/IP applied ensures the wiring simple and elegant. Figure.2 shows the software topology structure including the configuration files that insulate special devices and communication node topology from the reusable driving engine.

III. PROTOCOL DATA UNIT (PDU)

There are two types of PDUs:

i) PDU from MCU to Driving Engine

Each MCU sends UDP packet to the driving engine as follows:

struct *MCU2Engine* {

char *type*;//The type of the MCU: 0-sampling&driving;1-sampling;2-driving.

unsigned long *countReceivedPackets*;//The count of the driving packets

//received from the driving engine used to monitor the MCU status

//in the driving engine endpoint.

char *data*[];//The valid sample data according to the MCU.

};

This type of packet must be sent even the MCU is driving only. In this case the packet is regarded as a heartbeat packet.

ii) PDU from Driving Engine to MCU

The driving engine sends UDP packet to each driving MCU as follows:

struct *Engine2 MCU* {

char *data*[];//The driving data according to the MCU.

};

IV. TWO-DIMENSIONAL NOTATION

A. *Hardware item identifiers*

Each controlled hardware item on a device is identified in two-dimensional notation. An analogue output(such as a meter needle) is marked as *AO(i,j)*.An analogue input(such as a knob) is marked as *AI(i,j)*.An digital output(such as a light with two statuses: ON or OFF) is marked *DO(i,j)*. An digit input(such as a switch) is marked as *DI(i,j)*. Where *i* is the number identifying the panel that the item on and *j* is the number identifying the item.

B. *Hardware item reference in program code*

Each hardware item can be sampled or drove in a C/C++ program as follows:

i) Driving hardware items

AO(i,j)=x;// *AO(i,j)* such as a meter needle is drove to value *x*.

DO(i,j)=0;// *DO(i,j)* such as a light is turned off.

ii) Sampling hardware items

float *y=AI(i,j);//AI(i,j)* such as a knob is sampled and is saved in variable *y*.

int *s=DI(i,j)*;// *DI(i,j)* such as a switch is sampled and is saved in variable *s*.

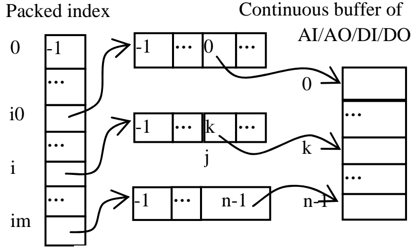

Figure 3. Packed index

C. Notation implementation

Figure.3 shows how to locate the hardware items in its buffer through the packed index. The hardware items lay on the panels from *i0* to *im*. The packed index is built based on the files *AO.cfg*, *AI.cfg*, *DO.cfg* and *DI.cfg*. These files must match the following syntax:

ItemList: Item ItemList Item: AItem|DItem
AItem: AO(i,j)[:Type]|AI(i,j)[:Type]
DItem: DO(i,j)[:State.]|DI(i,j)[:States]
Type: float| double States: State..State
State: number Based Zero
i:numberBasedZero
j:numberBasedZero

TABLE I. PRECISION LIST INTERPOLATION TABLE

x_i	x_0	x_1	…	x_K
y_i	y_0	y_1	…	y_K

V. PRECISION LISTS

A. Analogue output precision list

The drive process is as follows when a MCU controls an instrument needle (analogue output) to indicate scale *x*: *z=cvt(x)* → *z to MCU.*

where, the conversion calculation function *cvt(x)* is determined by hardware drive mode. The common problem is the error in terms of instrument indication instead of correction indication to *x*. To correct the error, the drive process can be modified to be: *y=f(x)* → *z=cvt(y)* → *z to MCU.*

That is, first to get *y* by means of proper turning of *x* through the function *f* and then *y* is used for driving. The function *f* can be materialized by means of interpolation based on the precision table (see Table I)

B. Analogue input precision list

Sample of current value *y* of an analogue input (such as an knob) can be described by: *receiving sample value z* → *y=cvt(z).* In the same way, the function *cvt(z)* is determined by hardware drive mode. Just like analogue output, sample of an analogue input is characterized by error, to correct which, sampling process can be improved as:

Receive sampling value from MCU z → *x=cvt(z)* → *y=f(x).*

The function *f* can be materialized by interpolation based on the precision table (see Table 1).

C. Precision lists on hard disk

If hardware is maintained, the precision table files will be modified and the program code must not need to be modified. Thus the precision lists must be stored in external hard disk. For every item *AO(i,j)* , its precision list is stored in the file *AO(i,j).prc* and for every *AO(i,j)* , the list is stored in the file *AI(i,j).prc* (see Fig.2). These precision lists will be loaded to memory by the driving engine in initializing phrase. The precision lists in memory can be accessed by the interface *PrecisionLists*:

struct *PrecisionLists*{
virtual double *AO*(int *i*, int *j*, double *x*);
//Return the interpolated value of *AO(i,j)* according to *x.*
virtual double *AI*(int *i*, int j, double *x*);
//Return the interpolated value of *AI(i,j)* according to *x.*
};

This interface will be passed to the MCU DLLs(Dynamic Link Library) in order to enable the DLLs accessing the precision lists.

VI. MCU DLLS

According to each MCU, a DLL exporting two functions *Sample ()* and *Drive()* as follows must be provided:

void *Sample*(char *data*[],//The *data* in embedded MCU format.
AcuracyLists **DoItp*// The interface to access the precision lists.
){
Step1: (Get *AI(i,j)* by parsing *data*) *x= GetData(data).*
Step2: (Do precision operation and update AI) *AI(i,j)= DoItp->AI(i,j,x).*
Step3: (Parse *data* then update *DI(i,j)*) *DI(i,j)= GetData(data).*
}

void *Drive*(char *data*[],//The data in embedded MCU format.
AcuracyLists **DoItp*// The interface to access the precision lists.
){
Step1: (Do precision operation on *AO(i,j*)) *x = DoItp->AO(i,j, AO(i,j));*
Step2: (Set *data* with *x*) *SetData(data,x).*
Step3: (Set *data* with *DI(i,j)*) *SetData(data, DI(i,j)).*
}

VII. THE DRIVING ENGINE

A. Configuration file of lower communication (lowercomm.cfg)

The lower communication is refers to the driving engine communicates with the MCUs based on TCP/IP. The file defines a list of items that each item as follows:

<McuName, McuIP, McuPort, DrvEngineIP, DrvEnginePort>.

The driving engine receives sample packets from *<DrvEngineIP, DrvEnginePort>* and sends driving packets from *<DrvEngineIP, DrvEnginePort>* to *<McuIP, McuPort>*.

In the driving engine endpoint, the DLL must be named as *McuName.dll* so that the driving engine can find the proper functions *Sample()* and *Drive()* .

B. *Configuration file of upper communication (uppercomm.cfg)*

The upper communication is refers to the driving engine communicates with the clients of the MCU data through a network of TCP/IP and(or) reflective memory boards. The file defines a list of items that each item as follows:

< DrvEngineIP, DrvEnginePort, AppIP, AppPort>.

< DrvEngineRfmCardID,DrvEngineRfmAddr, AppRfmCardID,AppRfmAddr>.

The driving engine receives driving packets from

< DrvEngineIP, DrvEnginePort > and(or)

< DrvEngineRfmCardID,DrvEngineRfmAddr >

and sends sample packets from

<DrvEngineIP, DrvEnginePort> to *<AppIP, AppPort>* and(or) writes them in *< AppRfmCardID, AppRfmAddr>*.

C. *Driving algorithm*

The key steps of the algorithm of the driving engine are as follows:

Step1: (Load files to initialize) Load *AO.cfg, AIcfg , DO.cfg* , *DI.cfg*,

AO(i,j).prc, AI(i,j).prc, LowerComm.cfg, UpperComm.cfg)

Step2: (Load DLL then find *Sample()* &*Drive()*)

FOR each *item* in *LowerComm.cfg* DO

Step2.1: (Load DLL) Call *LoadLibrary()*[7].

Step2.2: (Find *Sample()*&*Drive()*) Call *GetProcAddress()*[7].

Step3: (Loop) DO Step4 to Step5 UNTIL exit.

Step4: (Lower communication) FOR each *item* in *LowerComm.cfg* DO

Step4.1: (Receive from MCU then call *Sample()*) IF Receive from

< DrvEngineIP, DrvEnginePort> ok THEN Call *Sample()*.

Step4.2: (Call *Drive()*) Call *Drive()*.

Step4.3: (Send to MCU) Send the driving packet from

< DrvEngineIP, DrvEnginePort> to*<McuIP, McuPort>*.

Step5: (Upper communication) FOR each *item* in *UpperComm.cfg* DO

Step5.1: (Receive AOs/DOs) Receive *AO(i,j)/ DO(i,j)* from

<DrvEngineIP, DrvEnginePort> and(or)

<DrvEngineRfmCardID, DrvEngineRfmAddr>.

Step5.2: (Send AIs/DIs) Send *AI(i,j)/DI(i,j)* from

<DrvEngineIP, DrvEnginePort> to *< AppIP, AppPort >* and(or)

< AppRfmCardID, AppRfmAddr>.

VIII. CONCLUSIONS

It is a better way to apply embedded TCP/IP technology to sample and drive devices in man-in-loop simulators with a mass of hardware items. This method has advantages as follows:

i) The system wiring is simple and elegant than AD cards usually used.

ii) The maintenance is easy due to each device controlled by its MCU inside.

iii) The driving engine is a reusable software architecture and any number of MCUs can be integrated seamlessly in DLLs without any need to modify the architecture.

iv) The development is more efficient. To develop a new device control system, the only tasks are to implement the MCU embedded routines that must be done whenever any development methods applied and the MCU DLLs and then integrate these DLLs with the architecture. In addition, the two-dimensional notation makes hardware developers and software developers developing parallel.

v) The external precision lists make the precision maintenance valid.

vi) The external lower and upper communication configuration files make the communication performing without any communication code to be developed.

vii) The number of the driving engines bases on the mass of the devices and the real-time ability. For most cases, one driving engine with one or more NICs is enough.

viii) All clients of the device data communicate with the driving engine but not any MCUs and that make the upper communication simple.

ix) The system perform run-time and reliable.

Obviously, this method can be used in hardware-related systems especially in man-in-loop simulators to make the developing efficiently.

REFERENCES

[1] LIU Xin-shun, YAN Jian-guo, Data communication and acquisition of mode calculation computer for UAV's hardware-in-loop simulation in VxWorks.*Modern Electronics Technique*, 35(01),pp.7-9, 2012.

[2] XU Hai,CUI Lianhu,XU Guangyao, Research on realtime collection method of timing information in RTX environment. *Ship Electronic Engineering*,32(4),pp.59-61,2012.

[3] Jim Ledin, *Simulation Engineering: Build Better Embedded Systems Faster*. CMP books, San Francisco 2003.

[4] Buck Graham, TCP/IP addressing : designing and optimizing your IP addressing scheme.Morgan Kaufmann, San Diego, Calif. 2001.

[5] Michael J. Donahoo, Kenneth L. Calvert.,TCP/IP sockets in C : practical guide for programmers, 2nd Edition.Morgan Kaufmann Publishers, San Francisco 2009.

[6] SUN Yahong,Windows-based ditributed real-time simulation system. Electronic Science and Technology,25(03),pp.7-9, 2012.

[7] Jeffrey Richter, Christophe Nasarre, Windows Via C/C++. Microsoft Press, Washington 2011.