Linux 双进程应用示例

一、 概述

一台典型的工控设备通常包括若干通讯接口（网络、串口、CAN 等），以及若干数字 IO、AD 通道等。运行于设备核心平台的应用程序通过操作这些接口，实现特定的功能。通常为了高校高精度完成整个通讯控制流程，应用程序采用 C/C++语言来编写。图 1 表现了典型工控设备的组成关系。

图 1 典型工控设备框图

工控设备的另一个特点是鉴于设备大多是 24 小时连续运行，且无人值守，所以基本的工控设备是无显示的。英创的工控主板 ESM6800、ESM335x 等都大量的应用于这类无头工控设备之中。

在实际应用中，部分客户需要基于已有的无头工控设备，增加显示界面功能，以满足新的应用需求。显然保持已有的基本工控处理程序不变，通过相对独立的技术手段来实现显示功能，最符合客户的利益诉求。为此我们发展了一种双进程的程序设计方案来满足客户的这一需求。该方案的第一个进程，以客户已有的用 C/C++写的基础工控进程为基础，仅增加一个面向本地 IP（127.0.0.1）的侦听线程，用于向显示进程提供必要的运行工况数据。图 2 为增添了服务线程的工控进程：

图 2 带有侦听线程的基础工控进程

方案的第二个进程则主要用于实现显示界面，可以采用各种手段来实现，本文中介绍了使用 Qt 的 QML 语言加通讯插件的界面设计方法。第二个进程（具体是通讯插件单元）通过本地 IP，以客户端方式与基础工控进程进行 Socket 通讯，完成进程间数据交换。显示进程以及与工控进程的关系如图 3 所示：

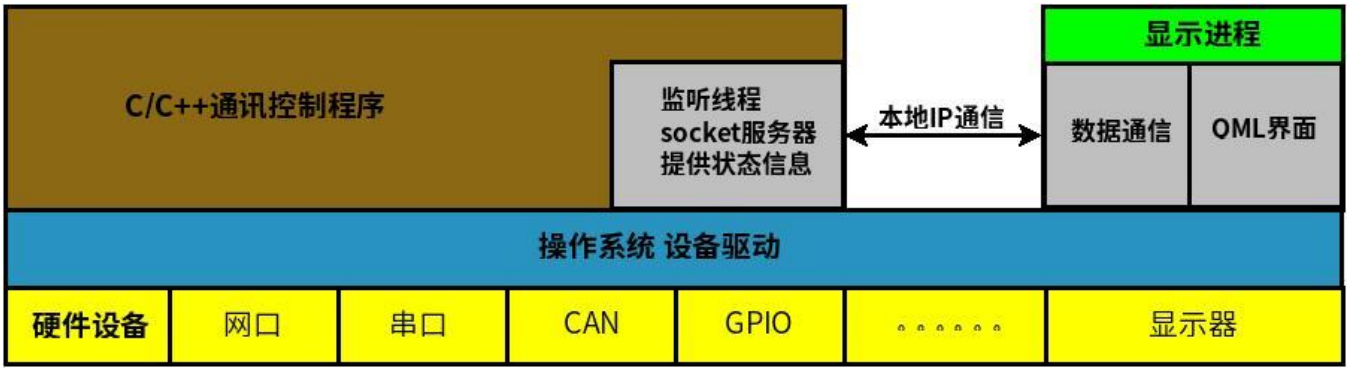
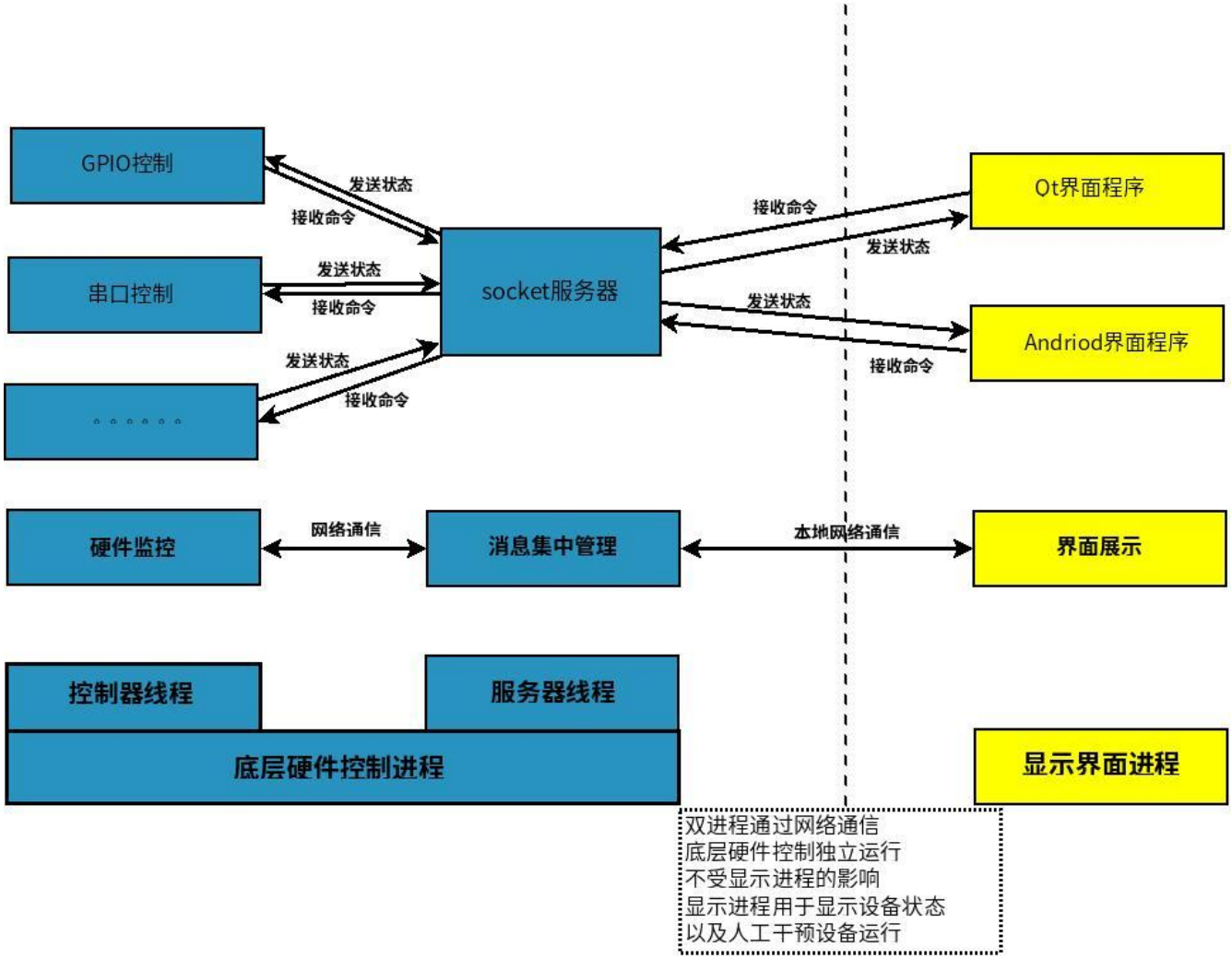


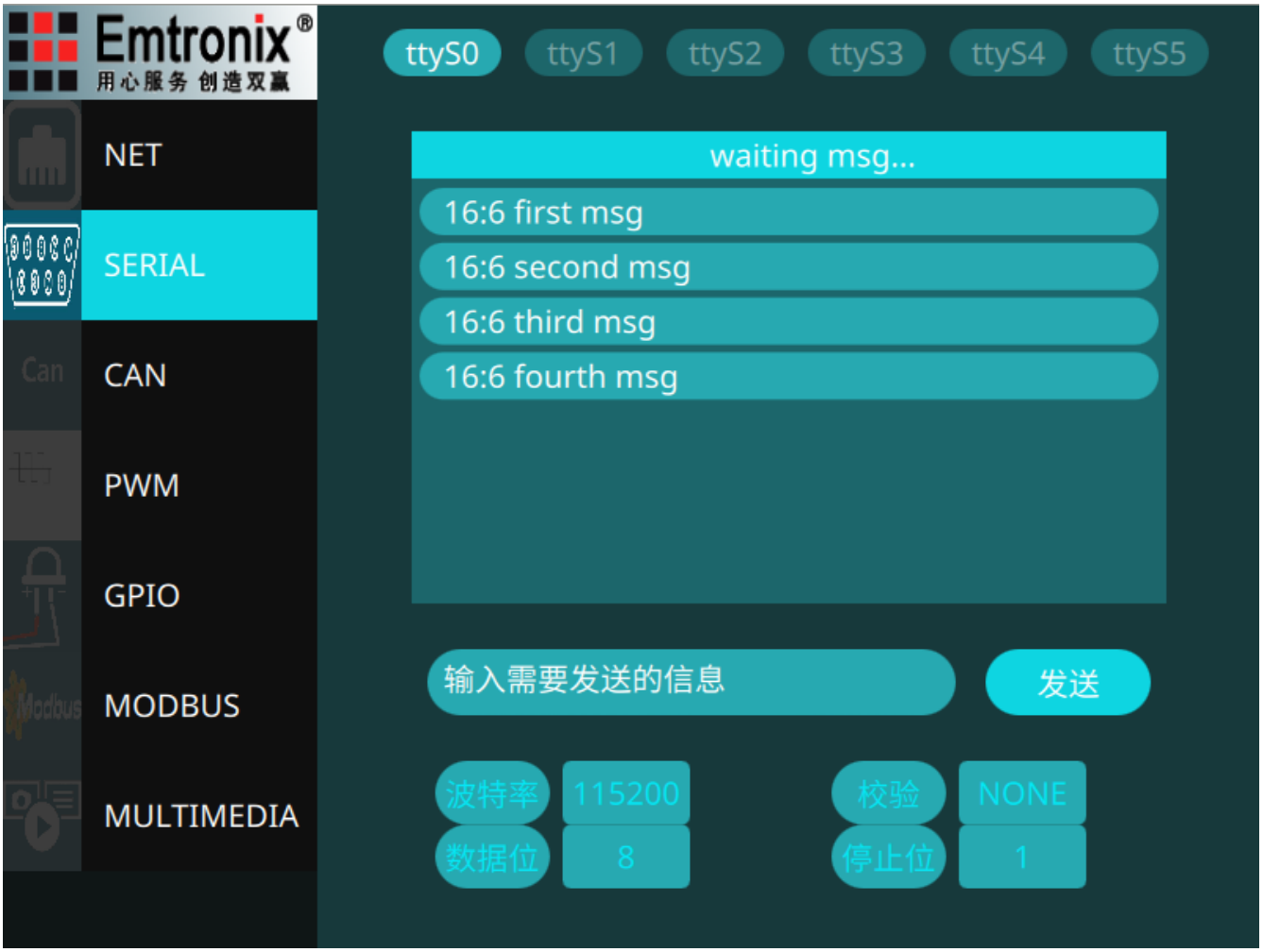
图 3 显示进程与工控进程

二、 系统设计

鉴于工业控制领域对系统运行的稳定性要求，控制系统更加倾向于将底层硬件控制部分与上层界面显示分开，两部分以双进程的形式各自独立运行。底层硬件控制部分将会监控系统硬件，管理外设等，同时收集系统的状态；而上层界面显示部分主要用于显示系统状态，并实现少量的系统控制功能，方便维护人员查看系统运行状态并且根据当前状态进行系统的调整。由于显示界面不一定是所有设备都配置，而且显示部分的程序更加复杂，从而更容易出现程序运行时的错误，将控制与显示分开能够避免由于显示部分的程序问题而影响到整个控制系统的运行，而且没有配置显示屏的设备也可以直接运行底层的控制程序，增加了系统程序的兼容性。显示与控制分离后，由于显示界面程序不需要处理底层硬件的管理控制，在设计时可以更加注重于界面的美化，而且界面程序可以采用不同的编程语言进行开发，比如使用 Qt C++或者 Android java，本文将介绍基于 Linux + Qt 的双进程示例程序供客户在实际开发中参考，关于 Android 程序请参考我们官网的另一篇文章。



如上图所示。整个系统分为控制和显示两个进程，底层硬件控制部分可以独立运行，使用多线程管理不同的硬件设备，监控硬件状态，将状态发送给 socket 服务器，并且从 socket 服务器接收命令来更改设备状态。Socket 服务器也是一个独立的线程，通过本地网络通信集中处理来自硬件控制线程以及显示程序的消息。显示界面需要连接上 socket 服务器才能正确的显示设备的状态，同时提供必须的人工控制接口，供设备使用过程中人为调整设备运行状态。目前在 ESM6802 工控主板上，界面程序可以采用 Qt C++编写，也可以使用 Android java 进行开发，本文仅介绍采用 Qt 的界面程序。显示程序界面用 QML 搭建，与底层通信的部分用独立的 Qt QML 插件实现，这样显示部分进一部分离为数据处理和界面开发，使得界面设计可以更加快捷。程序的整体界面效果如下图所示：



目前我们只提供了串口（SERIAL）和 GPIO 两部分的例程。下面将集中介绍程序中通过本地 IP 实现两个进程通信的部分供客户在实际开发中参考。

三、 控制端 C 程序

控制端程序主要分为两个部分，一个部分用于控制具体的硬件运行（下文称为控制器），另一个部分为 socket 服务器，用于与显示程序之间进行通信。由于本方案主要是为了展示在已有控制程序的基础上，增加显示界面功能，以满足新的应用需求，所以我们在此重点介绍在已有控制程序中加入 socket 服务器的部分，不再详细介绍各硬件的具体控制的实现。

增加本地 IP 通信的功能，首先需要在控制进程中新加入一个 socket 服务器线程，用

于消息的集中管理，实现底层硬件与上层的界面程序的信息交换，socket 服务器线程运行的函数体代码如下：

```
static void *_init_server(void *param)
{
	int server_sockfd, client_sockfd;
	int server_len;
	struct sockaddr_in server_address;
	struct sockaddr_in client_address;

	server_sockfd = socket(AF_INET, SOCK_STREAM, 0);
	server_address.sin_family = AF_INET;
	server_address.sin_addr.s_addr = inet_addr("127.0.0.1");//通过本地 ip 通信
	server_address.sin_port = htons(9733);
	server_len = sizeof(server_address);
	bind(server_sockfd, (struct sockaddr *)&server_address, server_len);

	listen(server_sockfd, 5);

	int res;
	pthread_t client_thread;
	pthread_attr_t attr;
	char id[4];
	client_element *client_t;

	while(1)
	{
		if(!client_has_space(clients))
		{
			printf("to many client, wait for one to quit...\n");
			sleep(2);
			continue;
		}
		printf("server waiting\n");
		client_sockfd = accept(server_sockfd, (struct sockaddr *)&client_address, (socklen_t
*)&server_len);

		//get and save client id
		read(client_sockfd, &id, 4);
		if((id[0]!='I') && (id[1]!='D'))
		{
```

```
                printf("illegal client id, drop it\n");
                close(client_sockfd);
                continue;
            }

            client_t = accept_client(clients, id, client_sockfd);
            printf("client: %s connected\n", id);

            //create a new thread to handle this connection
            res = pthread_attr_init(&attr);
            if( res!=0 )
            {
                printf("Create attribute failed\n" );
            }
            // 设置线程绑定属性
            res = pthread_attr_setscope( &attr, PTHREAD_SCOPE_SYSTEM );
            // 设置线程分离属性
            res += pthread_attr_setdetachstate( &attr, PTHREAD_CREATE_DETACHED );
            if( res!=0 )
            {
                printf( "Setting attribute failed\n" );
            }

            res = pthread_create( &client_thread, &attr,
                        (void *(*) (void *))socked_thread_func, (void*)client_t );
            if( res!=0 )
            {
                close( client_sockfd );
                del_client(clients, client_sockfd);
                continue;
            }
            pthread_attr_destroy( &attr );
        }

}
```

此函数创建一个 socket 用于监听（listen）等待显示程序连接，当接受（accept）一个连接之后创建一个新的线程用于消息处理，主要用于维护 socket 连接的状态，解析消息的收发方，并将消息转送到对应的接收方，在显示程序建立连接之前或者连接断开之后，控制器发送的消息将不会进行发送了，而控制器依然在正常运行，用于处理消息的新线程如

下：

```
static void *socked_thread_func(void *p)
{
	client_element *client_p = (client_element *)p;
	printf("started socked_thread_func for client: %s\n", client_p->id);
	fd_set fdRead;
	int ret, lenth;
	struct timeval aTime;
	struct msg_head msg_h;
	char *buf = (char *)&msg_h;   //from:2 char to 2 char msglenth:1 int
	buf[0] = client_p->id[2];
	buf[1] = client_p->id[3];
	char msg[100];
	client_element *send_to;
	struct tcp_info info;
	int tcp_info_len=sizeof(info);
	while(1)
	{
		FD_ZERO(&fdRead);
		FD_SET(client_p->sockfd, &fdRead);

		aTime.tv_sec = 2;
		aTime.tv_usec = 0;

		getsockopt(client_p->sockfd, IPPROTO_TCP, TCP_INFO, &info, (socklen_t
*)&tcp_info_len);
		if(info.tcpi_state == 1)
		{
			//printf("$$$%d tcp connection established...\n", client_p->sockfd);
			;
		}
		else
		{
			printf("$$$%d tcp connection closed...\n", client_p->sockfd);
			break;
		}

		ret = select( client_p->sockfd+1,&fdRead,NULL,NULL,&aTime );

		if (ret > 0)
		{
```

```
                    //判断是否读事件
                    if (FD_ISSET(client_p->sockfd, &fdRead))
                    {
                            //data available, so get it!
                            lenth = read( client_p->sockfd, buf+2, 6 );
                            if( lenth != 6 )
                            {
                                    continue;
                            }
                            // 对接收的数据进行处理，这里为简单的数据转发
                            lenth = read(client_p->sockfd, msg, msg_h.lenth);
                            if(lenth == msg_h.lenth)
                            {
                                    send_to = find_client(clients, msg_h.to);
                                    //printf("try to send to client %s\n", msg_h.to);
                                    if(send_to == NULL)
                                    {
                                            printf("can't find target client\n");
                                            continue;
                                    }
                                    write(send_to->sockfd, &msg_h, sizeof(struct msg_head));
                                    write(send_to->sockfd, msg, lenth);
                            }
                            // 处理完毕
                    }
            }
    }
    close( client_p->sockfd );
    del_client(clients, client_p->sockfd);
    pthread_exit( NULL );

}
```

这里收到消息后就解析消息头，发送到指定的端口去（控制器或者显示进程），由于实际应用中 socket 传送数据可能存在分包的情况，客户需要自行定义消息的数据格式来保证数据的完整性，以及对数据进行更严格的验证。

另一方面对于已有的控制器来说，需要在原来的基础上进行修改，在主线程中与

socket 服务器建立连接：

```
 sockedfd = socket(AF_INET, SOCK_STREAM, 0);
address.sin_family = AF_INET;
address.sin_addr.s_addr = inet_addr("127.0.0.1");
address.sin_port = htons(9733);
len = sizeof(address);

do
{
        res = connect(sockedfd, (struct sockaddr *)&address, len);
        if(res == -1)
        {
                perror("oops: connect error");
        }
}while(res == -1);
write(sockedfd, "IDG1", 4);
printf("###connected to server\n");
```

然后建立两个线程分别处理数据（data_thread_func)和命令(command_thread_func),其中 data_thread_func 用于监听硬件状态，并且发送相应的状态消息给 socket 服务器，而 command_thread_func 用于监听 socket 服务器的消息等待命令，用于改变硬件运行状态，不需要界面带有控制功能的客户可以不实现 commad_thread_func。以 GPIO 控制器为例：

```
void *gpio_controller::data_thread_func(void* lparam)
{
        gpio_controller *pSer = (gpio_controller*)lparam;

        fd_set fdRead;
        int ret=0;
        struct timeval aTime;
        unsigned int pinstates = 0;
        struct msg_head buf_h;


        while( 1 )
        {
                FD_ZERO(&fdRead);
                FD_SET(pSer->interface_fd,&fdRead);
```

```
                aTime.tv_sec = 2;
                aTime.tv_usec = 0;

                //等待硬件消息，这里是 GPIO 状态改变
                ret = select( pSer->interface_fd+1,&fdRead,NULL,NULL,&aTime );

                if (ret < 0 )
                {
                        //关闭
                        perror("select wrong");
                        pSer->close_interface(pSer->interface_fd);
                        break;
                }

                else
                {
                        //select 超时或者 GPIO 状态发生了改变，读取 GPIO 状态，发送给 socket 服务器
                        pinstates = INPINS;
                        ret = GPIO_PinState(pSer->interface_fd, &pinstates);
                        if(ret < 0)
                        {
                                printf("GPIO_PinState::failed %d\n", ret);
                                break;
                        }
                        sprintf((char *)&buf_h.to[0], "D1");
                        buf_h.lenth = sizeof(pinstates);
                        write(pSer->sockedfd, (void *)&buf_h.to[0], 6);
                        write(pSer->sockedfd, (void *)&pinstates, sizeof(pinstates));
                }
        }
        printf( "ReceiveThreadFunc finished\n");
        pthread_exit( NULL );
}

void *gpio_controller::command_thread_func(void* lparam)
{
        gpio_controller *pSer = (gpio_controller*)lparam;

        fd_set fdRead;
        int ret, len;
        struct timeval aTime;
```

```
struct outcom{
        unsigned int outpin;
        unsigned int outstate;
};
struct outcom out;
struct msg_head buf_h;

while( 1 )
{
        FD_ZERO(&fdRead);
        FD_SET(pSer->sockedfd,&fdRead);

        aTime.tv_sec = 3;
        aTime.tv_usec = 300000;

        //等待 socket 服务器的消息
        ret = select( pSer->sockedfd+1,&fdRead,NULL,NULL,&aTime );
        if (ret < 0 )
        {
                //关闭
                pSer->close_interface(pSer->interface_fd);
                break;
        }

        if (ret > 0)
        {
                //判断是否读事件
                if (FD_ISSET(pSer->sockedfd,&fdRead))
                {
                        len = read(pSer->sockedfd, &buf_h, sizeof(buf_h));
                        //获取 socket 服务器发送的信息，进行解析
                        if(len != sizeof(struct outcom))
                        {
                                printf("###invalid command lenth: %d, terminate\n", len);
                        }
                        len = read(pSer->sockedfd, &out, buf_h.lenth);

                        //write command
                        switch(out.outstate)
                        {
                                case 0:
                                        GPIO_OutClear(pSer->interface_fd, out.outpin);
```

```
                                        if(ret < 0)
                                                printf("GPIO_OutClear::failed %d\n", ret);
                                        //printf("GPIO_OutClear::succeed %d\n", ret);
                                        break;
                                case 1:
                                        GPIO_OutSet(pSer->interface_fd, out.outpin);
                                        if(ret < 0)
                                                printf("GPIO_OutSet::failed %d\n", ret);
                                        //printf("GPIO_OutSet::succeed %d\n", ret);
                                        break;
                                default:
                                        printf("###wrong gpio state %d, no operation\n",
out.outstate);
                                        ret = -1;
                                        break;
                        }
                        if(ret < 0)
                                break;
                }

        }
    }

    printf( "ReceiveThreadFunc finished\n");
    pthread_exit( NULL );
}
```

这里两个函数主要任务都是处理数据，data_thread_func 使用 select 函数来等待输入 GPIO 的状态改变事件，如果有状态改变或者 select 等待超时都读取一次 GPIO 的状态，然后发送给 socket 服务器；command_thread_func 监听服务器的消息，收到消息后进行解析，然后根据消息来操作 GPIO 输出信号。

通过这两个函数便与 socket 服务器建立了消息沟通通道，而 socket 服务器会自动将数据转发到显示进程，这种实现可以使得对已有程序的改动降到很低的程度。实际实现中，可以在 socket 服务器中增加状态机等其他功能，记录硬件状态信息等。

四、 显示程序

显示部分我们采用 Qt 来搭建，主要分为 QML 搭建的界面以及 Qt c++编写的数据处理插件。QML 是 Qt 提供的一种描述性的脚本语言，类似于 css，可以在脚本里创建图形对象，并且支持各种图形特效，以及状态机等，同时又能跟 Qt 写的 C++代码进行方便的交互，使用起来非常方便。采用 QML 加插件的方式主要是为了将界面设计与程序逻辑解耦，一般的系统开发中界面设计的变动往往多于后台逻辑，因此采用 QML 加插件的方式将界面设计与逻辑分离有利于开发人员的分工，加速产品迭代速度，降低后期维护成本。而且 QML 解释性语言的特性使得其语法更加简单，可以将界面设计部分交给专业的设计人员开发，而不要求设计人员会 c++等编程语言。Qt 底层对 QML 做了优化，将会优先使用硬件图形加速器进行界面的渲染，也针对触摸屏应用做了优化，使用 QML 能够更简单快捷的搭建流畅、优美的界面。QML 也支持嵌入 Javascript 处理逻辑，但是底层逻辑处理使用 Qt C++编写插件，能够更好的控制数据结构，数据处理也更加高效，Qt 提供了多种方式将 C++数据类型导入 QML 脚本中，更多详细资料可以查看 Qt 官方的文档。由于篇幅原因，我们将在另外一篇文章中更详细的介绍 QML 及插件的实现，在此我们还是集中介绍 socket 消息处理部分。

本例程中数据处理插件的任务就是连接 socket 服务器，与服务器进行通信，接收消息进行解析然后提供给 QML 界面，以及从 QML 界面获取消息给 socket 服务器发送命令。插件中通过 socket 进行通信的部分代码如下：

```
void MsgClient::cServer(void* param)
{
    MsgClient *client = (MsgClient *)param;
    int ret;
    int len;
    struct sockaddr_in address;
```

```
int sockedfd = socket(AF_INET, SOCK_STREAM, 0);
printf("sockedfd: %d\n", sockedfd);
client->sockedfd = sockedfd;
address.sin_family = AF_INET;
address.sin_addr.s_addr = inet_addr("127.0.0.1");          //本地 IP 通信
address.sin_port = htons(9733);
len = sizeof(address);
do
{
    printf("Client: connecting...\n");
    ret = ::connect(sockedfd, (struct sockaddr *)&address, len); //建立连接
    if(ret == -1)
    {
        perror("oops: connect to server error");
    }
    sleep(2);
}while(ret == -1);
write(sockedfd, "IDD1", 4);
printf("Client: connected to server\n");
emit client->serverConnected();

fd_set fdRead;
struct timeval aTime;
char buf[100];
unsigned int pinstates;
struct msg_head buf_h;

while(!client->exit_flag)
{
    FD_ZERO(&fdRead);
    FD_SET(sockedfd, &fdRead);

    aTime.tv_sec = 3;
    aTime.tv_usec = 0;

    ret = select(sockedfd+1, &fdRead, NULL, NULL, &aTime); //等待消息

    if(ret < 0)
    {
        perror("someting wrong with select");
    }
    if(ret > 0)
    {
```

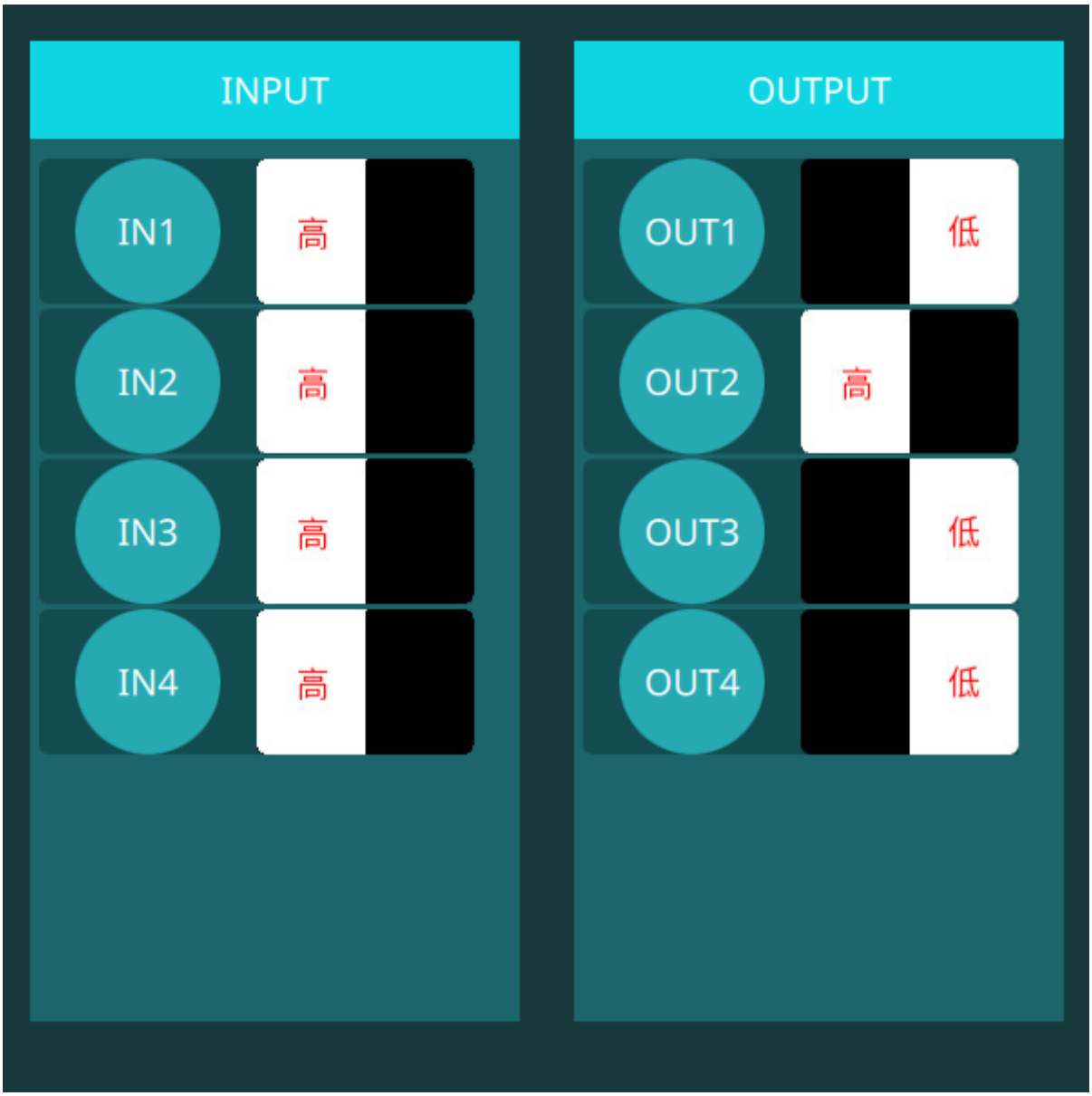
```
            if(FD_ISSET(sockedfd, &fdRead))
            {
                len = read(sockedfd, &buf_h, sizeof(buf_h));
                int i;
                switch (buf_h.from[0]) { //解析消息
                case 'S':
                        //串口信息
                    i = buf_h.from[1] - '0';
                    len = read(sockedfd, buf, buf_h.lenth);
                    client->rmsgQueue[i] << buf;
                    if(i == client->m_interface)
                        emit client->newMsgRcved();
                    memset(buf, 0, sizeof(buf));
                    break;
                case 'G':
                        //GPIO 信息
                    len = read(sockedfd, &pinstates, buf_h.lenth);
                    printf("get GPIO pinstates\n");
                    client->updateGPIOState(pinstates);
                    break;
                default:
                    break;
                }
            }
        }
    }

    close(sockedfd);
    pthread_exit(NULL);
}
```

如代码所示，插件首先通过本地 IP127.0.0.1 与 socket 服务器建立连接（connect），然后等待 socket 服务器的消息（select），收到消息后进行解析，判断是哪个硬件控制器发送的消息，然后更新相应的显示界面，这里的代码相对简单，只是为了展示通过本地 IP 实现显示进程与控制进程之间的通信，实际使用中客户需要对数据进行更严格的检验。

使用 QML 搭建串口控制界面如下图所示：


ttyS0
ttyS1
ttyS2
ttyS3
ttyS4
ttyS5
waiting msg...
15:11 first msg
15:11 second msg
15:11 third msg
15:11 fourth msg
输入需要发送的信息
发送
波特率
115200
校验
NONE
数据位
8
停止位
1

GPIO 控制器的显示效果如下：



由于篇幅原因，我们在此不详细介绍实现界面的 QML 脚本了，将会在另一篇文章中进行专门的介绍，感兴趣的用户可以关注我们官网上的文章更新，或者向我们要取程序源码。用户在实际开发中可以参考此方式实现显示进程与控制进程之间的通信，从而实现单独的显示进程，对已有的控制进程的更改控制到很小的程度，一方面减少了由于程序修改而造成控制程序的不稳定，另一方面使用 QML 又能快速的搭建界面，解决显示设备状态的需求。

五、 总结

实际测试过程中，我们在 ESM6802 工控板上运行本文介绍的程序，底层控制程序直接可以开机后台运行，显示程序开机后手动加载，通过本地 IP 地址与控制程序的 socket 服务器连接，然后实时更新系统状态，也能及时响应人工控制，如改变输出 GPIO 的输出状态，关掉显示程序之后，控制程序继续正常运行，之后还可以再次启动显示程序。

将底层控制与显示分开后，程序开发分工可以更加细致，也一定程度上增加了控制系统的稳定性，减小了维护成本。同时使用 QML 进行界面开发能够更加方便快速的更新系统的显示效果，完成产品迭代。由于底层控制与显示之间采用 socket 进行通信，显示部分也可以采用其他的开发环境，比如 ESM6802 也支持 Android 开发，用户在产品升级换代的时候就能够直接沿用底层控制部分的程序，而只对上层显示部分的程序进行调整。

有兴趣的客户可以和我们的工程师进行沟通获取更多信息以及程序代码。