固高科技
GOOGOL TECHNOLOGY

扩展功能

GXN 系列运动控制器编程手册

DMA 功能

R1.4

2021 年 8 月

版权申明

运动中的机器有危险！使用者有责任在机器中设计有效的出错处理和安全保护机制，固高科技没有义务或责任对由此造成的附带的或相应产生的损失负责。

联系我们

固高科技股份有限公司

地址：深圳市高新技术产业园南区深港产学研基地西座二楼 W211 室

电话：0755-26970817　26737236　26970824

传真：0755-26970821

电子邮件：googol@googoltech.com

网址：http://www.googoltech.com.cn

固高科技（海外）有限公司

地址：香港九龍觀塘偉業街 108 號絲寶國際大廈 10 樓 1008-09 室

電話：+(852) 2358-1033

傳真：+(852) 2719-8399

電子郵件：sales@googoltech.com

info@googoltech.com

網址：http://www.googoltech.com

臺灣固高科技股份有限公司

地址：台中市西屯區福中二路 10 巷 22 號 2 樓

電話：+886-4-23588245

傳真：+886-4-23586495

電子郵件：twinfo@googoltech.com

文档版本

版本号	修订日期
1.0	2018 年 10 月 16 日
1.1	2019 年 04 月 30 日
1.2	2020 年 01 月 11 日
1.3	2020 年 02 月 12 日
1.4	2021 年 03 月 18 日

前言

感谢选用固高运动控制器

为回报客户，我们将以品质一流的运动控制器、完善的售后服务、高效的技术支持，帮助您建立自己的控制系统。

固高产品的更多信息

固高科技的网址是 http://www.googoltech.com.cn。在我们的网页上可以得到更多关于公司和产品的信息，包括：公司简介、产品介绍、技术支持、产品最新发布等等。

您也可以通过电话（0755－26970817）咨询关于公司和产品的更多信息。

技术支持和售后服务

您可以通过以下途径获得我们的技术支持和售后服务：

电子邮件：support@googoltech.com；
电话：0755－26970843
发函至：深圳市高新技术产业园南区园深港产学研基地西座二楼 W211 室
固高科技股份有限公司
邮编： 518057

编程手册的用途

用户通过阅读本手册，能够了解运动控制器的功能，掌握函数的用法，熟悉编程实现。最终，用户可以根据自己特定的控制系统，编制用户应用程序，实现控制要求。

编程手册的使用对象

本编程手册适用于具有 C 语言编程基础或 Windows 环境下使用动态链接库的基础，同时具有一定运动控制工作经验，对伺服或步进控制的基本结构有一定了解的工程开发人员。

编程手册的主要内容

本手册由四章内容组成，详细介绍了运动控制器的 Gantry 功能及编程实现。

相关文件

关于控制器的调试和安装，请参见随产品配套的运动控制器用户手册。
关于控制器基本功能使用，请参见随产品配套的《GXN 系列运动控制器编程手册之基本功能》
关于更复杂的控制器功能，请参见随产品配套的《GXN 系列运动控制器编程手册之高级功能》
关于扩展模块的使用，请参见随产品配套的扩展模块编程手册。

产品相关手册及安装文件如驱动程序、dll 文件、例程、Demo 等，请登录固高科技公司网站下载，网址为：www.googoltech.com.cn/pro_view-53.html

目录

索引

1. 表格索引

2. 例程索引

3. 指令索引

一、 指令列表

提示	本章表格中右侧的数字为“页码”，其中指令右侧的为“**四、指令详细说明**”中的对应页码，其他为章节页码，均可以使用“超级链接”进行索引。 本手册中所有字体为蓝色的指令（如 GTN_CrdHsOn）均带有超级链接，点击可跳转至指令说明。

表 1 DMA 功能指令列表

指令	说明
GTN_CrdHsOn	打开插补 DMA
GTN_CrdHsOff	关闭插补 DMA
GTN_ScanHsOn	打开振镜 DMA（运控卡包含激光振镜功能时才可以使用）
GTN_ScanHsOff	关闭振镜 DMA
GTN_PosCompareHsOn	打开位置比较输出功能 DMA 通道
GTN_PosCompareHsOff	关闭位置比较输出功能 DMA 通道

二、 重点说明

1. DMA 功能为数据段批次压入运动控制器的功能，可以提高指令的传输速率。每次压入运动控制器的数据段数量由用户设定，默认为 200 段。开启 DMA 后，数据段会先被压入 DMA 的缓存区中，当数据段数到达阈值，如 200 段后，系统会自动将缓存区中的指令下发到运动控制器。压完所有的数据段之后，再循环调用 GTN_CrdData（振镜 DMA 功能需要调用 GTN_CrdDataEnd，位置比较 DMA 功能需要调用 GTN_PosCompareData 或 GTN_PosCompareData2D)指令将 DMA 缓存区中的剩余数据段(剩余的段数未达到阈值）压入运动控制器，直到指令返回值为 0。
2. 开启 DMA 功能后，插补运动功能的 GTN_CrdSpace、振镜功能的 GTN_ScanCrdSpace、位置比较输出功能的 GTN_PosCompareStatus 将查询的为 PC 端的剩余数据段空间（插补共有 4096 段空间，振镜共有 1000 段空间，位置比较共有 1000 段空间)。
3. 插补和振镜同时开启 DMA 功能时，GTN_CrdHsOn 和 GTN_ScanCrdHsOn 中的 link 参数应设为不同值。插补、振镜和位置比较输出等功能如果需要同时开启 DMA 功能时，需要选择不同的 DMA 缓存区序号。

三、 例程

例程 1 未使用前瞻的插补运动 DMA

```
int _tmain(int argc, _TCHAR* argv[])
{
    short i;
```

```
short rtn;
short core1;
long space;

core1 = 1;

rtn = GT_Open();
rtn = GT_Reset();

rtn = GT_LoadConfig("GTS800_test.cfg");
rtn = GT_ClrSts(1,4);
rtn = GT_ZeroPos(1,4);

TCrdPrm crdPrm;
rtn = GT_GetCrdPrm(1,&crdPrm);
crdPrm.dimension=2; // 坐标系为二维坐标系
crdPrm.synVelMax=500; // 最大合成速度：pulse/ms
crdPrm.synAccMax=1; // 最大加速度：pulse/ms^2
crdPrm.evenTime = 50; // 最小匀速时间：ms
crdPrm.profile[0] = 1; // 规划器对应到 X 轴
crdPrm.profile[1] = 2; // 规划器对应到 Y 轴

rtn = GT_SetCrdPrm(1, &crdPrm);//建立号坐标系，设置坐标系参数
rtn = GT_CrdClear(1, 0);//清除此缓存区

rtn = GT_CrdHsOn(1,0,1,200,0);

short crd=1;
double synVel=100.0,synAcc=1.0;

int count = 1000;
double x = 0;
double y = 0;
for(i=0;i<count;i++)
{
    x += 10;
    y += 20;
    rtn += GTN_LnXY(core1,crd,x,y,synVel,synAcc,0,0);
    if (rtn)
    {
        printf("crd 1 data error count=%d\n",count);
    }
}

rtn=GT_CrdStart(0x1,0);
```

```
    short run1;
    long seg;
    double pos[4];
    double targetVel;
    double crdSynVel;
    do
    {
        rtn = GT_CrdStatus(1,&run1,&seg,0);
        rtn = GT_GetPrfPos(1,&pos[0],2);
        printf("1pos=%lf,2pos=%lf\r",pos[0],pos[1]);
    }while(run1 == 1);

    getchar();

    return 0;
}
```

例程 2 使用新前瞻的插补运动 DMA

```
// 新前瞻初始化
void InitialNewLookAhead(short core,short crd,short fifo,int lookAheadNum)
{
    short rtn;
    EMachineMode machineMode;                   // 机床类型
    EVelSettingDef velDefineMode;               // 速度定义模式
    int axisLimitMode[8];                       // 轴限制模式
    EWorkLimitMode workLimitMode;               // 工件坐标系限制模式
    int axisFollowMode[8];                      // 轴跟随模式
    TLookAheadParameter lookAheadPara;          // 前瞻参数

    machineMode = NORMAL_THREE_AXIS;    // 标准三轴机床
    velDefineMode = NORMAL_DEF_VEL;     // 输入速度为三轴合成速度
    workLimitMode = WORK_LIMIT_VALID;   // 工件坐标系限制生效
    for (int i=0;i<8;++i)
    {
        axisLimitMode[i] = AXIS_LIMIT_NONE; // 轴限制不生效
        axisFollowMode[i] = 0;              // 非跟随轴
    }
    // 坐标系第4轴为跟随轴并限制轴运动能力
    axisLimitMode[3]=AXIS_LIMIT_MAX_VEL|AXIS_LIMIT_MAX_DV;// 轴最大速度和速度 最大
                                                          跳变生效
    axisFollowMode[3]=1;                // 跟随轴

    lookAheadPara.lookAheadNum = lookAheadNum;
    lookAheadPara.time = 1;             // 时间常数
    lookAheadPara.radiusRatio = 50;     // 曲率参数
```

```
    for (int i=0;i<8;++i)
    {
        lookAheadPara.vMax[i] = 5000;          // 轴最大速度限制
        lookAheadPara.aMax[i] = 100;           // 轴最大加速度限制
        lookAheadPara.DVMax[i] = 500;          // 轴跳变速度限制
        lookAheadPara.axisRelation[i] = i+1;   // 坐标系轴与前瞻轴一一映射
        lookAheadPara.scale[i] = 1000;         // 脉冲当量
    }

    rtn = GTN_SetupLookAheadCrd(core ,crd,machineMode);   // 设置机床模式
    rtn = GTN_SetAxisLimitModeLa(core ,crd,axisLimitMode);// 设置轴限制模式
    rtn = GTN_SetAxisVelValidModeLa(core ,crd,0xF);        // 设置轴速度有效，按位设置，0xF表示
                                                              前4个轴
    rtn = GTN_InitLookAheadEx(core ,crd,&lookAheadPara,fifo,0); // 设置前瞻参数（需要放在最后设
                                                                  置）
}

int _tmain(int argc, _TCHAR* argv[])
{
    short i;
    short rtn;
    short core = 1;
    rtn = GTN_Open();
    rtn = GTN_Reset(core);
    for (i=1;i<5;i++)
    {
        rtn = GTN_AlarmOff(core ,i);
        rtn = GTN_LmtsOff(core ,i);
    }
    rtn = GTN_ClrSts(core ,1,4);
    rtn = GTN_ZeroPos(core ,1,4);
    TCrdPrm crdPrm;
    rtn = GTN_GetCrdPrm(core ,1,&crdPrm);
    crdPrm.dimension=3;   // 坐标系为三维坐标系
    crdPrm.synVelMax=500; // 最大合成速度：500pulse/ms
    crdPrm.synAccMax=1;   // 最大加速度：1pulse/ms^2
    crdPrm.evenTime = 50;   // 最小匀速时间：50ms
    crdPrm.profile[0] = 1;    // 规划器1对应到X轴
    crdPrm.profile[1] = 2;    // 规划器2对应到Y轴
    crdPrm.profile[2] = 3;    // 规划器3对应到Z轴

    rtn = GTN_SetCrdPrm(core ,1, &crdPrm);// 建立1号坐标系，设置坐标系参数
    rtn = GTN_CrdClear(core ,1, 0);            // 清除此缓存区

    rtn = GT_CrdHsOn(1,0,1,200,0);
```

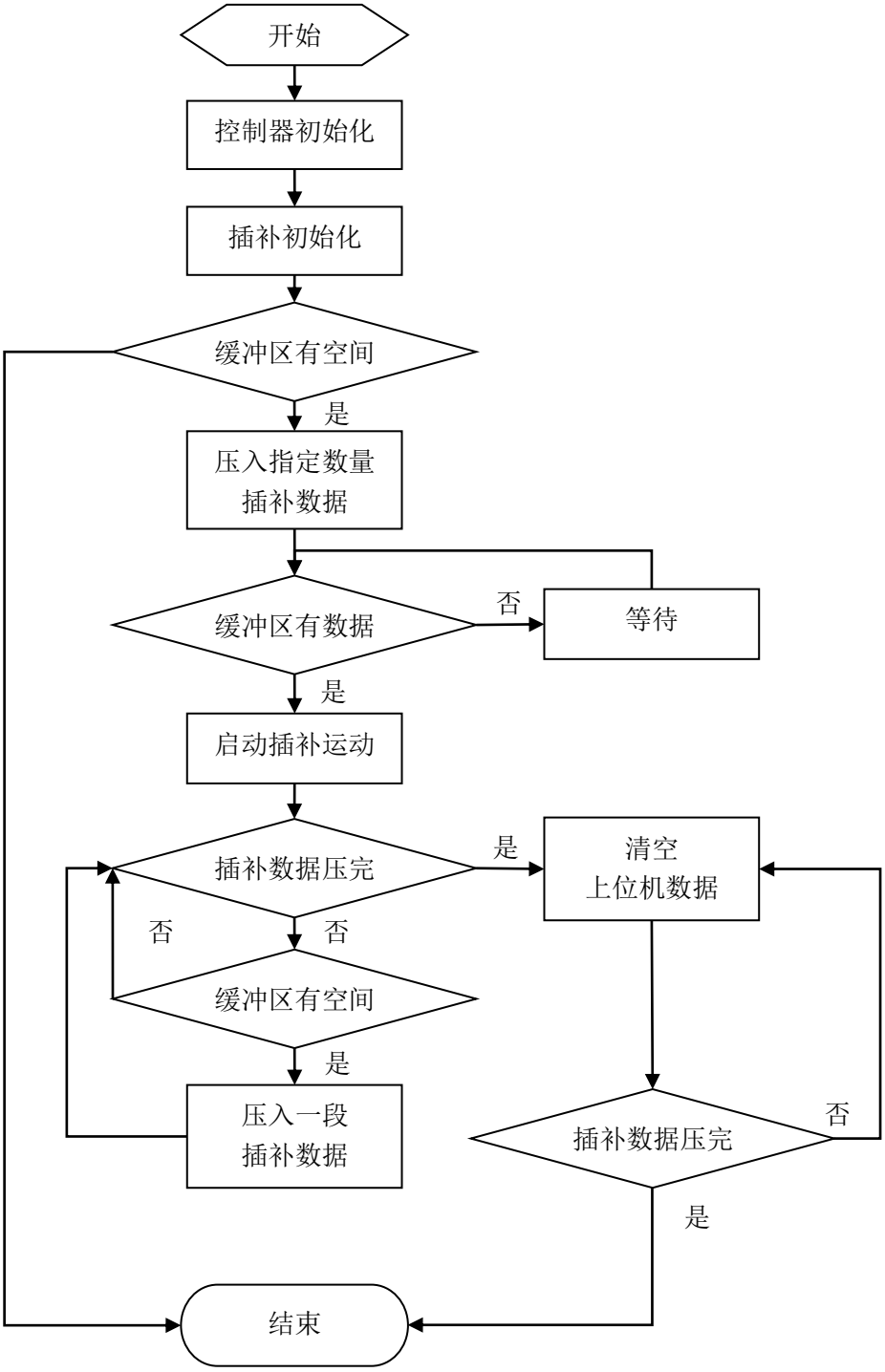
```
    double synVel=100.0*(1000/1000),synAcc=10*(1000000/1000);// 单位换算，此处分别为mm/s和
                                                              mm/s^2
    // 前瞻初始化
    InitialNewLookAhead(core ,crd,0,300);

    int count = 1000;
    double x = 0;
    double y = 0;
    for(i=0;i<count;i++)
    {
        x += 0.01;
        y += 0;
        rtn += GTN_LnXYZEx(core ,1,x,y,0,synVel,synAcc,0,0,0);
    }
    // 压入数据并启动运动
    while(1)
    {
        rtn=GTN_CrdDataEx(core ,1,NULL,0);
        if (rtn==0)
        {
            break;
        }
    }
    rtn=GTN_CrdStart(core ,1,0);
    return 0;
}
```

例程 3 实用化的插补例程

对于实际应用来说，加工的插补数据段会超过控制器缓冲区大小，不能一次压入到缓冲区，因此需要在一边执行插补运动过程中一边压入插补数据段。在刚开始，需要先向插补缓冲区压入一定数据量的插补数据，确认插补缓冲区已经有数据了再启动插补运动，然后在插补运动过程中判断插补缓冲区是否有空间，若有空间则继续压入插补数据。直到最后，当加工数据都通过插补指令压入完毕了，调用 GTN_CrdData/GTN_CrdDataEx 直到返回 0，把所有残存于上位机的数据压入控制器。调用 GTN_CrdData/GTN_CrdDataEx 直到返回 0 的过程中，不需要一直死循环等待，可以边执行其他操作，周期调用该指令直到返回 0 就不再调用。

下面给出流程图及执行一个加工文件的线程。完整例程请查看配套的“InterpolationTest”例程。




```
#define CRD_MAX (2)
#define STAGE_STATIC (0)
#define STAGE_AUTO_RUN (1)
#define STAGE_PUSH (2)
#define STAGE_END (3)

#define CRD_DATA_BUFFER_MAX (8000)
//全局变量
short rtn;
```

```
short core = 1;
short fifo = 0;
FILE *pgLogFile = NULL;
bool gThreadWork = FALSE;
long gTestLoop[CRD_MAX]; //循环测试次数
short gTestStage[CRD_MAX];
TGCodeInfo gCrdData[CRD_DATA_BUFFER_MAX];  //G代码转换成的插补数据
long gCrdDataCount = 0;         //读取的插补数据数量

void PushCrdData(short crd,TGCodeInfo *pCrdData)
{
double x;
double y;
double z;
double xCenter;
double yCenter;
short circleDir;
double synVel;
double synAcc = 10000;
long segNum;

xCenter = pCrdData->arc.center[0];
yCenter = pCrdData->arc.center[1];
synVel = pCrdData->vel;
segNum = pCrdData->seg;

switch(pCrdData->codeType)
{
case 1:
    x = pCrdData->line.endPos[0];
    y = pCrdData->line.endPos[1];
    z = pCrdData->line.endPos[2];
    rtn = GTN_LnXYZEx(core,crd,x,y,z,synVel,synAcc,segNum,0,0);
    if (rtn)
    {
        printf("crd:%d,loop:%ld,GTN_LnXYZEx=%d\n",crd,gTestLoop[crd-1],rtn);
    }
    break;
case 2:
    circleDir = 1;
case 3:
    circleDir = 0;
    x = pCrdData->arc.endPos[0];
    y = pCrdData->arc.endPos[1];
    z = pCrdData->arc.endPos[2];
    rtn = GTN_ArcXYCEx(core,crd,x,y,xCenter,yCenter,circleDir,synVel,synAcc,segNum,0,0);
```

```
        if (rtn)
        {
            printf("crd:%d,loop:%ld,GTN_ArcXYCEx=%d\n",crd,gTestLoop[crd-1],rtn);
        }
        break;

case 4:
        circleDir = 1;
case 5:
        circleDir = 0;
        x = pCrdData->arc.endPos[0];
        y = pCrdData->arc.endPos[1];
        z = pCrdData->arc.endPos[2];
        rtn = GTN_HelixXYCZEx(core,crd,x,y,z,xCenter,yCenter,0,synVel,synAcc,segNum,0,0);
        if (rtn)
        {
            printf("crd:%d,loop:%ld,GTN_HelixXYCZEx=%d\n",crd,gTestLoop[crd-1],rtn);
        }

        break;
default:
        printf("crd:%d,stage:%d,loop:%ld,unknowseg=%d\n",crd,gTestStage[crd-1],gTestLoop[crd-1],segNum);
        break;
}
}

DWORD WINAPI ThreadFunc(PVOID param)
{
long segIndex[CRD_MAX];          //当前传递哪一段数据
long i;
long pos;
double ratio;
TGCodeInfo *pCrdData;
short run;
long seg;
long space;
long count;
long remain;

for (i=0;i<CRD_MAX;i++)
{
        gTestStage[i] = 0;
        gTestLoop[i] = 0;
        segIndex[0] = 0;
        segIndex[1] = 0;
}
```

```
while (gThreadWork)
{
    for (short crd=1;crd <=CRD_MAX;crd++)
    {
        rtn = GTN_CrdSpace(core,crd,&space);
        rtn = GTN_CrdStatus(core,crd,&run,&seg);

        //刚开始先压入一部分数据，判断控制器有数据就启动运动
        if (STAGE_STATIC == gTestStage[crd-1])
        {
            //如果缓冲区有数据并且插补没有运动需要启动插补运动
            rtn = GTN_GetRemainderSegNum(core,crd,&remain,fifo);
            if ((0 != remain) && (0 == run))
            {
                //启动运动
                rtn = GTN_CrdStart(core,1 << (crd-1),0);
                if (rtn)
                {
                    printf("crd:%d,GTN_CrdStart=%d\n",crd,rtn);
                    continue;
                }

                gTestStage[crd-1] = STAGE_AUTO_RUN;
                gTestLoop[crd-1]++;
                printf("crd:%d,start and auto run\n",crd);
                printf("crd:%d test in %ld loop start...\n",crd,gTestLoop[crd-1]);
                continue;  //不再进行启动前压数据，进入执行压数据
            }

            //启动前压数据
            if (space > 0)
            {
                if (space > 1000)
                {
                    count = 1000;  //这里先压入段就启动
                }
                else
                {
                    count = space;
                }
            }
            else
            {
                printf("crd:%d,space is empty\n",crd);
                continue;
```

```
                }

                for (i=0;i<count;i++)
                {
                    pCrdData = &gCrdData[segIndex[crd-1]];
                    PushCrdData(crd,pCrdData);
                    segIndex[crd-1]++;
                }
            }
            //执行过程中，判断缓冲区有空间就压入数据
            else if(STAGE_AUTO_RUN == gTestStage[crd-1])
            {
                if (space > 0)
                {
                    pCrdData = &gCrdData[segIndex[crd-1]];
                    PushCrdData(crd,pCrdData);
                    segIndex[crd-1]++;
                }

                //执行完一次循环重新重头压入数据
                if (segIndex[crd-1] >= gCrdDataCount)
                {
                    segIndex[crd-1] = 0;
                    gTestStage[crd-1] = STAGE_PUSH;
                }
            }
            else if (STAGE_PUSH == gTestStage[crd-1])
            {
                rtn = GTN_CrdDataEx(core,crd,NULL,fifo);
                if (0 == rtn)
                {
                    //返回成功表示上面所有数据都被压入控制器缓冲区
                    gTestStage[crd-1] = STAGE_END;
                }
            }
            else if (STAGE_END)
            {
                //上一个循环的数据执行完毕，切换到下一次循环
                if (0 == run)
                {
                    gTestStage[crd-1] = STAGE_STATIC;
                }
            }
        }//for
    }//while
```

```
return 0;
}
```

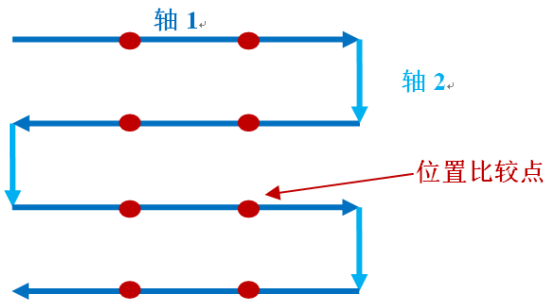
例程 4 振镜运动 DMA

```
short commandhandle(char * command,short returnValue)
{
    if(0!=returnValue)
    {
        printf("%s=%d\n",command,returnValue);
    }
    return 0;
}
int main(int argc,char *argv[])
{
    short rtn,crd,coreTemp;
    crd = 1;coreTemp=1;
    double jumpAcc,markAcc;
    jumpAcc = 0;markAcc = 0;
    short link = 1;
    bool scanStart = true;
    bool dataFlag = true;
    short space;
    TScanMap map;
    rtn = GTN_GetScanMap(coreTemp,crd,&map);
    commandhandle("GTN_GetScanMap",rtn);
    rtn = GTN_ScanInit(coreTemp,0,jumpAcc,markAcc,crd);
    commandhandle("GTN_ScanInit",rtn);
    rtn = GTN_ScanCrdClear(coreTemp,crd);
    commandhandle("GTN_ScanCrdClear",rtn);
    rtn = GTN_SetScanMode(coreTemp,FIFO_MODE_DYNAMIC,crd);
    commandhandle("GTN_SetScanMode",rtn);
    rtn = GTN_ScanHsOn(coreTemp,crd,link,100);
    commandhandle("GTN_ScanHsOn",rtn);
    while(1)
    {
        rtn = GTN_ScanCrdSpace(coreTemp,&space,crd);
        commandhandle("GTN_ScanCrdSpace",rtn);
        if((0 != space)&&(dataFlag == true))
        {
            segment = segment + 1;
            x = x + 2;
            y = y + 2;
            rtn = GTN_ScanJump(coreTemp,x,y,motionVel,crd);
            commandhandle("GTN_ScanJump",rtn);
            if(10000 == segment)
```

```
                {
                    dataFlag = false;
                    do
                    {
                        rtn = GTN_ScanCrdDataEnd(coreTemp,crd);
                    } while (rtn);
                }
            }
            if((scanStart == true)&&(10 == segment))
            {
                rtn = GTN_ScanCrdStart(coreTemp,crd); //  启动振镜运动
                commandhandle("GTN_ScanCrdStart",rtn);
                scanStart = false;
            }
            short pos[2];
            rtn = GTN_ScanGetCrdPos (coreTemp,&pos[0],crd);
            commandhandle("GTN_ScanGetCrdPos",rtn);
            printf("scan_X_Pos = %d      scan_Y_Pos = %d\r\n",pos[0],pos[1]);
        }
        return 0;
}
```

例程 5 位置比较 DMA

该例程实现轴 1 在 0~6000pulse 内往返的点位运动和轴 2 步长为 1000pulse 的单向点位运动，设置轴 1 的位置比较模式为 FIFO 模式，设置位置比较点为 2000pulse 和 4000pulse 的位置，并开启位置比较 DMA 功能。




```
#define POS_COMPARE_DATA_COUNT 1000  // 位置比较输出数据个数
#define X_POS 6000     // 点位运动轴1目标位置
#define Y_POS 1000     // 点位运动轴2目标位置
#define COMPARE_POS_1 2000        // 轴 1 位置比较点数据
#define COMPARE_POS_2 4000        // 轴 1 位置比较点数据

// 该函数检测某条GTN指令的执行结果，command为指令名称，error为指令执行返回值
void commandhandler(char *command, short error)
{
```

```
    // 如果指令执行返回值为非，说明指令执行错误，向屏幕输出错误结果
    if(error)
  {
        printf("\n%s = %d\n\n",command, error);
    }
}
int main (int argc, char* argv[])
{
    short sRtn;
    // 核号
    short core = 1;
    // 循环变量
    int i;
    // 位置比较输出索引号
    short index = 1;
    // 设置控制权的变量
    short permit;
    // 位置比较输出模式结构体
    TPosCompareMode mode;
    // 位置比较输出数据结构体
    TPosCompareData compareData;
    // 位置比较输出状态结构体
    TPosCompareStatus posCompareSts;
    // 位置比较输出位置
    long ComparePos[POS_COMPARE_DATA_COUNT + 1];
    // 点位运动状态
    long sts;
    // 点位运动规划器位置、编码器位置
    double prfPos, encPos;
    // 点位运动轴和轴目标位置
    long posTest[2];
    // 点位运动结构体
    TTrapPrm trap;
    // 打开运动控制器
    sRtn =GTN_Open(5, 2);
    // 指令返回值检测，请查阅例-1
    commandhandler("GTN_Open", sRtn);
    sRtn = GTN_Reset(core);
    commandhandler("GTN_Reset", sRtn);
    if(sRtn)
        return sRtn;

    sRtn = GTN_LoadConfig(core, "test.cfg");
    commandhandler("GTN_LoadConfig", sRtn);
    if(sRtn)
        return sRtn;
```

```
        sRtn = GTN_ClrSts(core, 1, 8);
        commandhandler("GTN_ClrSts", sRtn);
        if(sRtn)
            return sRtn;

        sRtn = GTN_ZeroPos(core, 1, 8);
        commandhandler("GTN_ZeroPos", sRtn);
        if(sRtn)
            return sRtn;

        // 使用脉冲计数器
        sRtn = GTN_EncOff(core, 1);
        commandhandler("GTN_ZeroPos", sRtn);
        if(sRtn)
            return sRtn;

        // 设置控制权
        short pPermit[16];

        // 读取硬件通道信号输出类型
        sRtn = GTN_GetTerminalPermitEx(core,
                                       1,              // 模块栈号
                                       MC_HSO,         // MC_HSO(18)高速io输出
                                       pPermit,  // 按位读取硬件输出通道信号输出类型
                                       1,              // 读取的起始硬件信号输出通道索引
                                       16);       // 读取的硬件信号输出通道个数
        commandhandler("GTN_GetTerminalPermitEx", sRtn);
        if(sRtn)
            return sRtn;

        // 设置控制权
        permit = 0x2;

        // 设置第一路HSO为第一路位置比较输出模式
        sRtn = GTN_SetTerminalPermitEx(core,
                                       1,              // 模块栈号
                                       MC_HSO,         // MC_HSO高速io输出
                                       &permit,        // 设置第一路位置比较输出有效bit1：第一路位置比较输出，bit2第二路位置比较输出，0：无效1：有效
                                       1,              // 设置起始硬件信号输出通道索引为
                                       1);             // 设置的硬件信号输出通道个数为
        commandhandler("GTN_SetTetminalPermitEx", sRtn);
        if(sRtn)
            return sRtn;
```

```
        // 设置第一路GPO为第一路位置比较输出模式
        sRtn = GTN_SetTerminalPermitEx(core,1, MC_GPO, &permit, 1, 1);
        commandhandler("GTN_SetTetminalPermitEx", sRtn);
        if(sRtn)
            return sRtn;

        // 停止位置比较输出
        sRtn = GTN_PosCompareStop(core, index);
        commandhandler("GTN_PosCompareStop", sRtn);
        if(sRtn)
            return sRtn;

        // 清空位置比较输出数据
        sRtn = GTN_PosCompareClear(core, index);
        commandhandler("GTN_PosCompareClear", sRtn);
        if(sRtn)
            return sRtn;

        // 读取位置比较输出模式
        sRtn = GTN_GetPosCompareMode(core, index, &mode);  // 位置比较索引
        commandhandler("GTN_GetPosCompareMode", sRtn);
        if(sRtn)
            return sRtn;

        // 设置位置比较输出模式为FIFO模式
        mode.mode = 0;                          // 0：FIFO模式1：Linear模式2：PSO立即模式3：PSO等待到位触发模式
        mode.dimension = 1;                     // 1:一维2:二维
        mode.sourceMode = 1;                // 位置比较源0：编码器1：脉冲计数器
        mode.sourceX = 1;                   // X轴比较源对应的实际轴为轴
        mode.outputMode = 0;                // 输出模式0：脉冲1：电平
        mode.outputPulseWidth = 10;         // 输出脉冲宽度，单位：us 电平模式下该参数无效
        mode.errorBand = 10;

        sRtn = GTN_SetPosCompareMode(core, index, &mode);
        commandhandler("GTN_SetPosCompareMode", sRtn);
        if(sRtn)
            return sRtn;


        // 开启DMA,设置阈值为
        // 如果指令数达到阈值，则自动将缓存区指令下发至运动控制器
        sRtn = GTN_PosCompareHsOn(core, index, 1, 200);
        commandhandler("GTN_PosCompareHsOn", sRtn);
        if(sRtn)
            return sRtn;
```

```
// 生成位置比较输出位置点[2000, 4000, 4000, 2000, 2000, 4000 ,4000,..., 4000]
for(i = 0; i<POS_COMPARE_DATA_COUNT; i++)
{
    if(i == 0) {
        ComparePos[i] = COMPARE_POS_1;
    }
    else{
        if(ComparePos[i-1] == COMPARE_POS_2) {
            ComparePos[i] = COMPARE_POS_1;
            ComparePos[i+1] = COMPARE_POS_1;
        }
        else{
            ComparePos[i] = COMPARE_POS_2;
            ComparePos[i+1] = COMPARE_POS_2;
        }
        i++;
    }
    if(i == POS_COMPARE_DATA_COUNT - 1) {
        ComparePos[i] = COMPARE_POS_2;
        break;
    }
}

// 设置FIFO模式位置比较输出数据
for(i = 0; i<POS_COMPARE_DATA_COUNT; i++)
{
    compareData.hso = 0x1; // 第一路HSO输出
    compareData.gpo = 0x1;// 第一路GPO输出
    compareData.segmentNumber = i;
    compareData.pos = ComparePos[i];

    sRtn = GTN_PosCompareData(core, index, &compareData);
    commandhandler("GTN_PosCompareData", sRtn);
    if(sRtn)
        return sRtn;
}
// 将剩余数据压入DSP缓存区
do
{
    sRtn = GTN_PosCompareData(core, index, NULL);
    if(sRtn != 1&&sRtn != 0)
    {
        commandhandler("GTN_PosCompareData", sRtn);
        return sRtn;
```

```
    }
}while(sRtn);

// 启动位置比较输出
sRtn = GTN_PosCompareStart(core, index);
commandhandler("GTN_PosCompareStart", sRtn);
if(sRtn)
    return sRtn;

// 启动点位运动
// 轴和轴伺服使能
sRtn =GTN_AxisOn(core, 1);
commandhandler("GTN_AxisOn", sRtn);

sRtn =GTN_AxisOn(core, 2);
commandhandler("GTN_AxisOn", sRtn);

// 位置清零
sRtn = GTN_ZeroPos(core, 1, 8);
commandhandler("GTN_ZeroPos", sRtn);

// 将轴和轴设为点位运动模式
sRtn = GTN_PrfTrap(core, 1);
commandhandler("GTN_PrfTrap", sRtn);

sRtn = GTN_PrfTrap(core, 2);
commandhandler("GTN_PrfTrap", sRtn);

// 读取点位运动参数(需要读取所有运动参数到上位机变量)
sRtn = GTN_GetTrapPrm(core, 1, &trap);
commandhandler("GTN_GetTrapPrm", sRtn);

// 设置需要修改的运动参数
trap.acc = 10;              // 单位：pulse/ms^2
trap.dec = 10;              // 单位：pulse/ms^2
trap.smoothTime = 0;    // 自带的软件滤波

// 设置轴点位运动参数
sRtn = GTN_SetTrapPrm(core, 1, &trap);
commandhandler("GTN_SetTrapPrm", sRtn);

// 设置轴点位运动参数
sRtn = GTN_SetTrapPrm(core, 2, &trap);
commandhandler("GTN_SetTrapPrm", sRtn);

// 设置轴的目标速度
```

```
        sRtn = GTN_SetVel(core, 1, 10);          // 单位：pulse/ms
        commandhandler("GTN_SetVel", sRtn);

        // 设置轴的目标速度
        sRtn = GTN_SetVel(core, 2, 10);          // 单位：pulse/ms
        commandhandler("GTN_SetVel", sRtn);

        posTest[0] = 0;
        posTest[1] = 0;

        for(i = 0; i<POS_COMPARE_DATA_COUNT/2; i++)
        {
            if(i%2 == 0) {
                posTest[0] = X_POS;
            }
            else{
                posTest[0] = 0;
            }
            posTest[1] += Y_POS;

            // 设置轴1的目标位置
            sRtn = GTN_SetPos(core, 1, posTest[0]);     // 单位：pulse
            commandhandler("GTN_SetPos", sRtn);

            // 启动轴1的运动
            sRtn = GTN_Update(core, 0x1);
            commandhandler("GTN_Update", sRtn);

            do
            {
                // 读取轴1的状态
                sRtn = GTN_GetSts(core, 1, &sts);
                // 读取轴1规划位置
                sRtn = GTN_GetPrfPos(core, 1, &prfPos);
                sRtn = GTN_GetEncPos(core, 1, &encPos);
                // 读取位置比较输出功能状态
                sRtn = GTN_PosCompareStatus(core, index, &posCompareSts);

                printf("segmemt = %-5ld Axis1:sts=0x%-10lx prfPos=%-10.1lf encPos = %-10.1lf
pulseCount = %-5ld\r", posCompareSts.segmentNumber, sts, prfPos, encPos, posCompareSts.pulseCount);
            }while(sts&0x400);// 等待AXIS轴规划停止
            printf("\n");

            // 设置轴2目标位置
            sRtn = GTN_SetPos(core, 2, posTest[1]);
```

```
            // 启动轴2的运动
            sRtn = GTN_Update(core, 0x2);
            commandhandler("GTN_Update", sRtn);

            do
            {
                // 读取轴2的状态
                sRtn = GTN_GetSts(core, 2, &sts);
                // 读取轴2规划位置
                sRtn = GTN_GetPrfPos(core, 2, &prfPos);
                // 读取位置比较输出功能状态
                sRtn = GTN_PosCompareStatus(core, index, &posCompareSts);

                printf("segmemt = %ld Axis2:sts=0x%-10lx prfPos=%-10.1lf pulseCount = %ld\r",
posCompareSts.segmentNumber, sts, prfPos, posCompareSts.pulseCount);
            }while(sts&0x400);// 等待AXIS轴规划停止
            printf("\n");
        }

        // 伺服关闭
        sRtn = GTN_AxisOff(core, 1);
        sRtn += GTN_AxisOff(core, 2);
        printf("\nGTN_AxisOff()=%d\n", sRtn);

        sRtn = GTN_Close();
        commandhandler("GTN_Close", sRtn);

        getch();
        return 0;
    }
```

四、 指令详细说明

指令 1 GTN_CrdHsOn

指令原型	short GTN_CrdHsOn(short core,short crd,short fifo,short dmaBuf =1,unsigned short threshold=200, short lookAheadInMc=0)		
指令说明	打开插补 DMA		
指令类型	立即指令，调用后立即生效。	**章节页码**	6
指令参数	该指令共有 6 个参数，参数的详细信息如下。		
core	核号，正整数，取值范围[1,2]		
crd	坐标系号，取值范围[1,2]		
fifo	插补坐标系的FIFO号，取值范围[0,1]		

dmaBuf	默认取 1，取值范围[1,4]，如果同时使用振镜 DMA，则振镜 DMA 指令对应的 link 应该为不同于插补 DMA 指令的 dmaBuf
threshold	阈值，PC 机指令达到该阈值时自动启动 DMA 发送数据。
lookAheadlnMc	默认取 0

指令 2　GTN_CrdHsOff

指令原型	short GTN_CrdHsOff(short core,short crd,short fifo);		
指令说明	关闭插补 DMA		
指令类型	立即指令，调用后立即生效。	**章节页码**	6
指令参数	该指令共有 3 个参数，参数的详细信息如下。		
core	核号，正整数，取值范围[1,2]		
crd	坐标系号，取值范围[1,2]		
fifo	插补坐标系的FIFO号，取值范围[0,1]		

指令 3　GTN_ScanHsOn

指令原型	short GTN_ScanHsOn(short core,short scan=1,short dmaBuf =1,unsigned short threshold=200);		
指令说明	打开振镜DMA		
指令类型	立即指令，调用后立即生效。	**章节页码**	6
指令参数	该指令共有 4 个参数，参数的详细信息如下。		
core	核号，正整数，取值范围[1,2]		
scan	振镜坐标系号，取值范围[1,4]		
dmaBuf	默认取 1，取值范围[1,4]，如果同时使用插补 DMA，则插补 DMA 指令对应的 link 应该为不同于振镜 DMA 指令的 dmaBuf		
threashold	阈值，PC机指令达到该阈值时自动启动DMA发送数据。		

指令 4　GTN_ScanHsOff

指令原型	short GTN_ScanHsOff(short core,short scan);		
指令说明	关闭振镜DMA		
指令类型	立即指令，调用后立即生效。	**章节页码**	6
指令参数	该指令共有 2 个参数，参数的详细信息如下。		
core	核号，正整数，取值范围[1,2]		
scan	振镜坐标系号，取值范围[1,4]		

指令 5　GTN_PosCompareHsOn

指令原型	short GTN_PosCompareHsOn(short core, short index=1, short dmaBuf=1, unsigned short threshold=200)		
指令说明	打开位置比较输出的DMA通道。		
指令类型	立即指令，调用后立即生效。	**章节页码**	6
指令参数	该指令共有 3 个参数，参数的详细信息如下。		
core	核号，正整数，取值范围[1,2]		
index	缓存区 FIFO，正整数，取值范围请参照《GXN 系列运动控制器编程手册之基本功能》表		

	13-1 中的“位置比较输出”一栏
dmaBuf	DMA 数据缓存区序号，正整数，默认值为 1。取值范围请参照《GXN 系列运动控制器编程手册之基本功能》中的表 13-1 中的“DMA 数据缓存区序号”一栏。 如果指令数达到该阈值，系统会自动将缓存区指令下发至运动控制器。

指令 6　GTN_PosCompareHsOff

指令原型	short GTN_ScanHsOff(short core, short index=1)		
指令说明	关闭位置比较输出的DMA通道。		
指令类型	立即指令，调用后立即生效。	**章节页码**	6
指令参数	该指令共有 2 个参数，参数的详细信息如下。		
core	核号，正整数，取值范围[1,2]		
index	缓存区 FIFO，正整数，取值范围请参照《GXN 系列运动控制器编程手册之基本功能》表 13-1 中的“位置比较输出”一栏		