```
/*********************************************************************
 * Source file             : Convert.cpp
 * File summary            : Conversion processing main file
 * Created by              :
 * Updated on (created on) : 2002.10.01(2002.10.01)
 * Remarks                 : Compile switches for compiling are listed below.
 * HISTORY                 :
 * ID  -- DATE -- ----- NOTE ----------------------------------
 * 00  2002.10.01 Created
 *********************************************************************/
#ifndef _CONVERT_
#define _CONVERT_

// ---------------------------------------------------------------------------
// Include
// ---------------------------------------------------------------------------
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <time.h>
#include <iostream>
#include <algorithm>
#include <map>
#include <set>
#include <string>
#include <vector>
#include <cstdio>
using namespace std;


#define _STL_HAS_NAMESPACES
#include <unistd.h>

#pragma hdrstop

//---------------------------------------
// Environmental switch
//---------------------------------------
#pragma package(smart_init)


//---------------------------------------------------------------
// Structure for 1-line data save
//---------------------------------------------------------------
typedef struct stCsvLineData{
    vector<string>  word;
    vector<int>     type;
}stCsvLineData;

//---------------------------------------------------------------
// CSV data class declaration
//---------------------------------------------------------------
class CCsvFile
{

public:
    CCsvFile();                                 // Constructor
    virtual ~   CCsvFile();                             // Destructor

    bool ReadFile(string filename, string delm=" ," );  // File read
    bool SetAtRec(int index );                          // Record pointer move

    bool SetDataStr(string str);                        // 1-line data setting
    bool SetDataStr(char *wkstr, string delm);          // 1-line data setting
    bool DeleteData(void);                              // 1-line data deletion
    string Strings(int index);                          // 1-line data character string fetch
    int  GetDataCnt(void);                              // 1-line data counter

protected:
    void DataClear(void);                               // Memory clear processing

    vector<stCsvLineData> vCsvLineData;                 // Read and stored data
    vector<stCsvLineData>::iterator p_vCsvLineData;     // Read and stored reference pointer
};

//---------------------------------------------------------------------------
```

```
//--------------------------------
// Table for analysis processing
//--------------------------------
typedef struct stCalculateData{
   double  fTimes;                                  // Accumulated time (msec)[for analysis]
                                                    //                  (sec) [for read]
   int     nTimeFlg;                                // t1-t6 flag
   int     nGear;                                   // Gear
   int     nGearTime;                               // Gear determination time duration (msec)
   int     nCalcTime;                               // Required time
   int     nPtnReadFlg;                             // Pattern read flag 1: Read data exists.
   int     nWriteFlg;                               // Location to write pattern file result
   bool    bIdle;                                   // Idle state IDLE=true
   bool    bClutch;                                 // Clutch state ON  =true
   int     nFlag;                                   // Holds same flag due to use of previous version
                                                    //  0:IDLE
                                                    //  1: Start
                                                    //  2: Constant speed
                                                    //  3: Stop processing (clutch disengaged)
                                                    //  4: Decelerate (including shift-down)
                                                    //  5: Accelerate (including shift-up)
   double  fV;                                      // Speed (for pattern data read)
   double  fT;                                      // Time (for pattern data read)
   double  fA;                                      // Acceleration (km/sec)
   double  fCarA;                                   // Acceleration (m/msec)
   double  fbtwnTime;                               // Time required between basic points (for pattern data read)
   double  fVref_sp;                                // Reference speed
   double  fVana_sp;                                // Analysis vehicle speed
   double  fNegrevo;                                // Engine speed
   double  fTe;                                     // Engine torque
   double  fF;                                      // Driving force
   double  fRL;                                     // R/L rolling resistance
}stCalculateData;

//--------------------------------
// Excess force ratio data
//--------------------------------
typedef struct stExceedForce{
   int     nGear;                                   // Gear position
   double     fGearti;                              // Gear ratio
   double     fForcePer;                            // Transmission efficiency
   double     fFreePer;                             // Excess ratio
   double     fMinPer;                              // Lower-limit engine speed (%-normalized)
   double     fMinNe;                               // Lower-limit engine speed (calculated)
}stExceedForce;

typedef struct stTeFree{
   double fTeFree[20];
   double fNe[20];
   double fMaxNe[20];
   double fF[20];
}stTeFree;

//-----------------------------------------------------------------
// Max. torque data
//-----------------------------------------------------------------
struct MAX_TORQUE
{
   double fEgtq;                        // Engine torque
   double fEgrevo;                      // Engine speed
};


//=========================================================================
// Constant declaration (define)
//=========================================================================
#define MAIN_ENVFILE                    "DATA"
#define DEF_FORCE_ON98                  0.98                    // 98%
#define DEF_FORCE_OFF95                 0.95                    // 95%
// For GVW margin determination
#define DEF_FORCEOVER                   (8000.0)                // GVW determination 8t
#define DEF_FORCE_OV_GEAR2              2.0                     // Excess ratio 2.0 (8t or more)
#define DEF_FORCE_OV_GEAR3              1.7                     // Excess ratio 1.7 (8t or more)
#define DEF_FORCE_OV_GEAR4              1.3                     // Excess ratio 1.3 (8t or more)
#define DEF_FORCE_UN_GEAR2              2.4                     // Excess ratio 2.4 (less than 8t)
#define DEF_FORCE_UN_GEAR3              1.7                     // Excess ratio 1.7 (less than 8t)
#define DEF_FORCE_UN_GEAR4              1.6                     // Excess ratio 1.6(less than 8t)
```

```
// GVW normalized engine speed
#define DEF_FORCE_NE_GEAR2          5                   // Normalized engine speed 5%
#define DEF_FORCE_NE_GEAR3          11                  // _______11%
#define DEF_FORCE_NE_GEAR4          19                  // _______19%
#define DEF_FORCE_NE_GEAR5          26                  // _______26%

// Output data (header portion)
#define DEF_PRINT_POS1              "time(s)"
#define DEF_PRINT_POS2              "Vtarget(km/h)"
#define DEF_PRINT_POS3              "Vreal(km/h)"
#define DEF_PRINT_POS4              "Ne(rpm)"
#define DEF_PRINT_POS5              "Te(N-m)"
#define DEF_PRINT_POS6              "N_norm(%)"
#define DEF_PRINT_POS7              "T_norm(%)"
#define DEF_PRINT_POS8              "Shift"


#define DEF_MAXGEAR     (7)                         // Max. gear
#define DEF_MAXDIFFER   10.0                        // Vehicle speed difference
#define POINTS_MAX      16                          // Max. points available for processing
#define RESULTMAX       90                          // Number of points to be calculated
//---------------------------
// Output message
//---------------------------
#define MSG_WRITE_FILE_ERROR        "File write error."     // File write error


#define BFSZ                        1024
#define NG                          -1
#define OK                          1

#define MAX_GEAR                    20   // Max. gear count


//=========================================================================
// Class declaration
//=========================================================================
class TCalculateProc
{
public:
    TCalculateProc();                           // Constructor
    virtual ~  TCalculateProc();                // Destructor

  // Function declaration
  bool   Init();                                // Analysis processing initialization
  bool   Init(string FileName);                 // Analysis processing initialization (with file name designated)
  bool   DataClear();                           // Analysis data clear processing
  bool   EnvRead();                             // Environmental data read processing
  bool   PtnRead();                             // Pattern file read processing
  bool   PtnRead(string szFile);                // Pattern file read processing (with file name designated)


  int   CalculateProcess();                     // Initiates analysis processing.
  void   GetCalculateDataFileName(string &szFile);      // Obtains analysis file name.
  //-------------------------------
  // Frequently used functions
  //-------------------------------
  void   BtwnTimeSet(map<double, stCalculateData>::iterator p_first,
            map<double, stCalculateData>::iterator p_end );     // Sets section time for specified section.
  void   BtwnCarASet(map<double, stCalculateData>::iterator p_first,
            map<double, stCalculateData>::iterator p_end );     // Sets acceleration for specified section.

  //---------------------------------
  // Analysis processing steps
  //---------------------------------
  bool   Calculate_progress1();                 // Calculates reference vehicle speed and reference acceleration.
  bool   Calculate_progress2();                 // Determines flag for pattern compatible with previous version.
  bool   Calculate_progress3();                 // Sets gear (according to engine speed).

  //-------------------------------
  // Section setting functions
  //-------------------------------
  bool   Calculate_T1T2Set();                   // Search and section setting for start (t1,t2)
  bool   Calculate_Start_Following(
          map<double,stCalculateData>::iterator &p_first );     // Processing until analysis vehicle speed reaches reference vehicle speed

  bool   Calculate_T3Set(map<double,stCalculateData>::iterator p_first,
```

```
            map<double,stCalculateData>::iterator &p_second );      // Section setting for acceleration (t3)
    bool   Calculate_T3Check(map<double,stCalculateData>::iterator p_first,
                map<double, stCalculateData>::iterator p_second,
                int tmpGear, int OrgGear );                     // Running capability check for section setting
    bool   Calculate_GearUp(map<double,stCalculateData>::iterator p_first,
                map<double, stCalculateData>::iterator p_second,
                int &tmpGear );                             // Shift-up until stationary engine speed is exceeded
    bool   Calculate_NeGear(map<double,stCalculateData>::iterator p_first,
                int &nGear );                               // Shift-down until stationary engine speed is exceeded
    bool   Calculate_T6Set(map<double,stCalculateData>::iterator p_first,
            map<double, stCalculateData>::iterator p_second );      // Section setting for deceleration (t6)

    bool   Calculate_SetIDLE(
            map<double,stCalculateData>::iterator p_first,
            map<double,stCalculateData>::iterator p_second );       // IDLE section processing
    bool   Calculate_Set_Start(
            map<double,stCalculateData>::iterator p_first,
            map<double,stCalculateData>::iterator p_second );       // Start section gear setup processing
    bool   Calculate_Set_SteadyState(
            map<double,stCalculateData>::iterator p_first,
            map<double,stCalculateData>::iterator p_second );       // Constant speed section gear setup processing
    bool   Calculate_Set_Deceleration(
            map<double,stCalculateData>::iterator p_first,
            map<double,stCalculateData>::iterator p_second );       // Deceleration section gear setup processing
    bool   Calculate_Set_Acceleration(
            map<double,stCalculateData>::iterator p_first,
            map<double,stCalculateData>::iterator p_second );       // Acceleration section gear setup processing

    void   DispCalculateData();
    int    WriteAllCalculateData();

private:
    string  m_OutputData;                                   // Output file name
    //***************System parameter setting***************
    double  m_fPAI;                                         // Circle circumference ratio to diameter
    //Analysis parameter
    double  m_fUnitTime;                                    // Analysis interval (sec)
    int     m_nMaxGear;                                     // Max. number of gears
    //-----------------------------------------------------
    // Vehicle information setting
    double  m_fCarMaxW;                                     // Max. payload (Kg)
    double  m_fCarIniW;                                     // Empty vehicle mass (kg)
    double  m_fPersons;                                     // Riding capacity
    double  m_fPersonW;                                     // Weight per person
    double  m_fCarMe;                                       // Empty vehicle weight of car (kN)
    double  m_fCarMc;                                       // Payload of car (kN)
    double  m_fPersonM;                                     // Weight of riding capacity (kN)
    double  m_fEFact;                                       // Inertial weight ratio equivalent in rotation section (E_FACT)
    double  m_fMFact;                                       // Inertial weight ratio equivalent in rotation section (M_FACT)

    double  m_fTarR;                                        // Tire rolling radius data
    double  m_fOverHeight;                                  // Overall vehicle height
    double  m_fOverWidth;                                   // Overall vehicle width
    //-----------------------------------------------------
    double  m_fClutch_Release;                              // Clutch release normalized engine speed (%)
    double  m_fClutch_Meet;                                 // Clutch meet normalized engine speed (%)
    double  m_fClutch_ReleaseNe;                            // Clutch release engine speed
    double  m_fClutch_MeetNe;                               // Clutch meet engine speed
    //-----------------------------------------------------
    // Engine specifications setting
    double  m_fStandardOutputRotation;                      // Rated output engine speed [rpm]
    double  m_fOutputRotation;                              // Loaded limit engine speed [rpm]
    double  m_fStandardTorque;                              // Rated torque
    double  m_fIdleNe;                                      // Idling (IDLE) engine speed
    //-----------------------------------------------------

    //-----------------------------------------------------
    // Gear setting
    vector<double> m_fGearHi;                               // Gear ratio
    double  m_fLastReduceGear;                              // Final reduction ratio
    double  m_fUd;                                          // Final reduction ratio (transmission efficiency)
    //-----------------------------------------------------

    //-----------------------------------------------------
    // Starting condition initial value
    int     m_nInitGear;                                    // Starting gear initial value
    //-----------------------------------------------------
```

```
    // [Gearshift condition initial value]
    double  m_fTg;                                      // Gear hold time (tg:sec)
    //-----------------------------------------------------

    //***************System parameter setting***************
    //-----------------------------------------------------
    // Other member variables
    double  m_fFixedNe;                                 // Max. engine speed, rated engine speed
    double  m_fKg;                                      // Gravitational acceleration
    //-----------------------------------------------------

private:
    //-------------------------------------
    // Processing used in initialization
    //-------------------------------------
    double GetMaxNe();                                  // Max. engine speed setting

    bool   GetKG(double &fKg);                          // Gravitational acceleration setting
    bool   GetExceedF();                                // Excess force ratio data read processing
    double GetGearPass( int nGear );                    // Obtains gear transmission efficiency.

    int    ReadMaxTorqueData(string FileName);          // Sets max. torque data table.
    double   GetLineReviseMaxTorque(double fNe);
                                                        // Obtains max. torque data, and executes calculation.
    bool   CalcForceWithNeSet(int nGear, double Te, double fNe,
                              double &fF );             // Sets driving force.

    bool   CheckForce(int nGear, double fVana,
                      double fCarA );                   // Determines shift-up availability.
    bool   CalcTeMaxSp(int nGear, double fTm,
                double fVbef, double &fV, double &fNe );// Target-speed follow processing

    //--------------
    // Calculation logic
    //--------------
    double   CalcRL(double fV);                         // Calculates rolling resistance.
    double   GetCarWeight(bool bFlag, int nGear=0);     // Reads and calculates vehicle body weight.
    int   GetGearHi(int nGrear,
                    double &fGearHi);                   // Obtains gear ratio.
    int   GetGeariN(int nGrear,
                    double &fGearti);                   // Obtains gear ratio.
    int   GetNe( int nGear,double fVg,
                    double &fNe);                       // Obtains engine speed.
    int   GetV( int nGear,double fNe,
                    double &fVg);                       // Obtains speed.
    int   GetTe( int nGear,double fV,double fA,
                double nNe, double &fTe);               // Calculates torque.
    int   GetTe_NotRevise( int nGear,double fV,double fA,
                double nNe, double &fTe);               // Calculates torque (no correction).

    //------------------------
    // Spline complement relationship
    //------------------------
    //--------------------------
    // Analysis file related
    //--------------------------
    int    WriteHead(FILE *fp);                         // HED file write processing

public:
    map<double,stCalculateData> setCalculateData;       // Analysis data table
    map<double,stCalculateData>::iterator p_setCalculateData;   // Analysis data pointer

private:
    set<MAX_TORQUE>      m_MaxTorque;                   // Max. torque data
    set<MAX_TORQUE>::iterator p_MaxTorque;              // _____ pointer
    set<stExceedForce> m_ExceedForce;                   // Excess force ratio table
    set<stExceedForce>::iterator p_ExceedForce;         // _____ pointer

    //------------------------------
    // Max. torque data operator
    //------------------------------
    friend bool operator<(const MAX_TORQUE& a, const MAX_TORQUE& b ){
      // Uniquely sorted by engine speed.
      return( a.fEgrevo < b.fEgrevo );
    };
    //------------------------------
    // Data operator for excess force ratio
```

```
    //------------------------------
    friend bool operator<(const stExceedForce& a, const stExceedForce& b ){
        // Uniquely sorted by gear.
        return( a.nGear < b.nGear );
    };

};

//============================================================================
// Class declaration
//============================================================================
class TCommFun
{
public:
    TCommFun();
private:

public:
    bool    AStrToDouble(string szData,
                         double &fData);

    void    Trim(string &str );
    bool    FileExists( string filename );
private :

public :
      virtual ~  TCommFun();

};

//----------------------------------------------------------------------
// Class declaration
//----------------------------------------------------------------------
TCalculateProc *CalculateProc;
TCommFun *CommFun;
//---------------------------------------------------------------------------
/**/
/*********************************************************************
 * Function name           : CCsvFile
 * Function summary        : Constructor
 * Explanation             : Class constructor
 *                         :
 * Argument (input)        : None
 * Argument (output)       : None
 * Argument (I/O)          : None
 * Return value            : None
 * Created by              :
 * Updated on (created on) :
 * Remarks                 :
 *********************************************************************/
CCsvFile::CCsvFile()
{

    return;
}

/**/
/*********************************************************************
 * Function name           : ~  CCsvFile
 * Function summary        : Destructor
 * Explanation             : Class destructor
 *                         :
 * Argument (input)        : None
 * Argument (output)       : None
 * Argument (I/O)          : None
 * Return value            : None
 * Created by              :
 * Updated on (created on) :
 * Remarks                 :
 *********************************************************************/
CCsvFile::~  CCsvFile()
{
    //------------------------------
    // Clear memory.
    //------------------------------
    DataClear();
```

```
    return;
}

/**/
/*********************************************************************
 * Function name          : DataClear
 * Function summary       : Memory clear processing
 * Explanation            : Memory data is cleared.
 *                        :
 * Argument (input)       : None
 * Argument (output)      : None
 * Argument (I/O)         : None
 * Return value           : None
 * Created by             :
 * Updated on (created on) :
 * Remarks                :
 *********************************************************************/
void CCsvFile::DataClear(void)
{
    //-----------------------------------------------------
    // Read file memory area deletion processing
    //-----------------------------------------------------
    if( vCsvLineData.empty() != true ){
        for( p_vCsvLineData = vCsvLineData.begin();
                p_vCsvLineData != vCsvLineData.end();
                p_vCsvLineData++ ){                         // Loops for number of file read lines.
            if( p_vCsvLineData->word.empty() != true ){             // In case of 1-line data
                p_vCsvLineData->word.erase( p_vCsvLineData->word.begin(),
                                            p_vCsvLineData->word.end() );
                p_vCsvLineData->word.clear();                       // Clears 1-line data.
            }
            if( p_vCsvLineData->type.empty() != true ){             // In case of 1-line data
                p_vCsvLineData->type.erase( p_vCsvLineData->type.begin(),
                                            p_vCsvLineData->type.end() );
                p_vCsvLineData->type.clear();                       // Clears 1-line data.
            }
        }
        vCsvLineData.erase( vCsvLineData.begin(),
                            vCsvLineData.end() );               // Deletes 1 line.
        vCsvLineData.clear();                                   // Deletes container.
    }

    return;
}
/**/
/*********************************************************************
 * Function name          : ReadFile
 * Function summary       : File read
 * Explanation            : File is read and set in internal data.
 *                        :
 * Argument (input)       : strFileName : File name
 * Argument (input)       : delm        : Delimiter
 * Argument (output)      : None
 * Argument (I/O)         : None
 * Return value           : true : Normal    false : Failure
 * Created by             :
 * Updated on (created on) :
 * Remarks                :
 *********************************************************************/
bool CCsvFile::ReadFile(string strFileName, string delm )
{
    string tmpStr;                                          // 1-line read character string buffer
    FILE *fp;
    char *p;
    char wkstr[4096];

    // Reads and opens file.
    fp = fopen( strFileName.c_str(), "r" );

    if((fp == NULL)||(ferror(fp))) return false;                    // File open failure

    //----------------------------
    // Internal data clear
    //----------------------------
    DataClear();

    //----------------------------
```

```
    // File read
    //----------------------------
    while( !feof(fp) ){
        memset( wkstr, 0x00, sizeof( wkstr ));
        p = fgets( wkstr, 4096, fp );                    // 1-line read
        if( p == NULL ) break;
        if( ferror(fp) ) break;

        // 1-line data setting
        tmpStr = string( wkstr );
        if(( delm == " ," )||( delm == ", )){
            SetDataStr(tmpStr);
        }else{
            SetDataStr( &wkstr[0], delm);
        }
    }
    fclose(fp);

    return true;
}
/**/
/*********************************************************************
 * Function name          : SetAtRec
 * Function summary       : Record pointer move processing
 * Explanation            : File record pointer is moved.
 *                        :
 * Argument (input)       : index : Record
 * Argument (output)      : None
 * Argument (I/O)         : None
 * Return value           : true : Normal    false : Failure
 * Created by             :
 * Updated on (created on) :
 * Remarks                :
 *********************************************************************/
bool CCsvFile::SetAtRec(int index )
{
    int i;

    if(( (int)(vCsvLineData.size()) >= index )&&
       ( index > 0 )){
        for( p_vCsvLineData = vCsvLineData.begin(), i = 1;
            i != index; i++ ){
            p_vCsvLineData++;
        }
    }else{
        p_vCsvLineData = vCsvLineData.end();
        return false;
    }

    return true;
}
/**/
/*********************************************************************
 * Function name          : SetDataStr
 * Function summary       : 1-line data setup processing
 * Explanation            : 1-line data from file is set.
 *                        :
 * Argument (input)       : str : Set 1-line character string
 * Argument (output)      : None
 * Argument (I/O)         : None
 * Return value           : true : Normal    false : Failure
 * Created by             :
 * Updated on (created on) :
 * Remarks                :
 *********************************************************************/
bool CCsvFile::SetDataStr(string str)
{
    stCsvLineData tmpCsvLineData;
    char wkstr[256];                                    // Read character string buffer
    char wkstr_wd[256];
    int i;
    int tmp_delm;                                       // Flag indicating delimiting section
    int tmp_delmIndex;                                  // Delimiting start position
    string tmpStr;


    if( tmpCsvLineData.word.empty() != true ){
```

```
    tmpCsvLineData.word.erase( tmpCsvLineData.word.begin(),
                               tmpCsvLineData.word.end() );
    tmpCsvLineData.word.clear();
    tmpCsvLineData.type.erase( tmpCsvLineData.type.begin(),
                               tmpCsvLineData.type.end() );
    tmpCsvLineData.type.clear();
}
// Checks by obtaining characters one by one.
tmp_delm = 0;
tmp_delmIndex = 0;

sprintf( wkstr, "%s", str.c_str() );
for( i = 0; wkstr[i] != 0x00; i++ ){
    if( wkstr[i] == '"' ){
        // Delimiting section start?
        if( tmp_delm == 0 ){                        // Delimiting section start occurrence
            tmp_delm = 1;                           // Sets delimiting position.
            i++;
            tmp_delmIndex = i;                      // Sets delimiting start position.
        }else{                                      // Delimiting section end?
            memset( wkstr_wd, 0x00, sizeof( wkstr_wd ));
            memcpy( wkstr_wd, &wkstr[tmp_delmIndex],
                              i - tmp_delmIndex );  // Sets delimiting position.
            tmpStr = string( wkstr_wd );
            tmpCsvLineData.word.push_back( tmpStr );
            tmpCsvLineData.type.push_back( tmp_delm );
            tmp_delm = 0;
            // Sets next delimiting start position.
            i++;
            tmp_delmIndex = i;
            while(1){
                if( wkstr[i] == 0x00 ){
                    i--;
                    break;
                }
                if(( wkstr[i] == ' ' )||
                   ( wkstr[i] == '\t' )||
                   ( wkstr[i] == '\r' )||
                   ( wkstr[i] == '\n' )||
                   ( wkstr[i] == ',' )){
                    i++;
                    tmp_delmIndex = i;
                }else{
                    i--;
                    break;
                }
            }
        }
    }else if(( wkstr[i] == ' ' )||
             ( wkstr[i] == '\t' )||
             ( wkstr[i] == '\r' )||
             ( wkstr[i] == '\n' )||
             ( wkstr[i] == ',' )){                  // Normal delimiting position found?
        if( tmp_delm == 1 ){                        // If character " has already appeared
            // the data is regarded as character data.
        }else{
            memset( wkstr_wd, 0x00, sizeof( wkstr_wd ));
            memcpy( wkstr_wd, &wkstr[tmp_delmIndex],
                              i - tmp_delmIndex );  // Sets delimiting position.
            tmpStr = string( wkstr_wd );
            tmpCsvLineData.word.push_back( tmpStr );
            tmpCsvLineData.type.push_back( tmp_delm );
            // Sets next delimiting start position.
            tmp_delmIndex = i;
            while(1){
                if( wkstr[i] == 0x00 ){
                    i--;
                    break;
                }
                if(( wkstr[i] == ' ' )||
                   ( wkstr[i] == '\t' )||
                   ( wkstr[i] == '\r' )||
                   ( wkstr[i] == '\n' )||
                   ( wkstr[i] == ',' )){
                    i++;
                    tmp_delmIndex = i;
                }else{
```

```
                    i--;
                    break;
                }
            }
        }
    }
    if(( wkstr[i-1] != ' ' )&&
      ( wkstr[i-1] != '\t' )&&
      ( wkstr[i-1] != '\r' )&&
      ( wkstr[i-1] != '\n' )&&
      ( wkstr[i-1] != ',' )){
       memset( wkstr_wd, 0x00, sizeof( wkstr_wd ));
       memcpy( wkstr_wd, &wkstr[tmp_delmIndex],
                        i - tmp_delmIndex );                    // Sets delimiting position.
       tmpStr = string( wkstr_wd );
       tmpCsvLineData.word.push_back( tmpStr );
       tmpCsvLineData.type.push_back( tmp_delm );
    }

    vCsvLineData.push_back( tmpCsvLineData );

    return true;
}
/**/
/*****************************************************************
 * Function name          : SetDataStr
 * Function summary       : 1-line data setup processing (with delimiter)
 * Explanation            : 1-line data from file is set.
 *                        :
 * Argument (input)       : wkstr : Set 1-line character string
 * Argument (input)       : delm  : Delimiter
 * Argument (output)      : None
 * Argument (I/O)         : None
 * Return value           : true : Normal    false : Failure
 * Created by             :
 * Updated on (created on) :
 * Remarks                :
 *****************************************************************/
bool CCsvFile::SetDataStr(char *wkstr, string delm)
{
    stCsvLineData tmpCsvLineData;
    char wkstr_wd[4096];
    int i;
    int tmp_delm;                                               // Flag indicating delimiting section
    int tmp_delmIndex;                                          // Delimiting start position
    string tmpStr;


    if( tmpCsvLineData.word.empty() != true ){
        tmpCsvLineData.word.erase( tmpCsvLineData.word.begin(),
                                   tmpCsvLineData.word.end() );
        tmpCsvLineData.word.clear();
        tmpCsvLineData.type.erase( tmpCsvLineData.type.begin(),
                                   tmpCsvLineData.type.end() );
        tmpCsvLineData.type.clear();
    }
    // Checks by obtaining characters one by one.
    tmp_delm = 0;
    tmp_delmIndex = 0;

    for( i = 0; wkstr[i] != 0x00; i++ ){
        if( wkstr[i] == '"' ){
            // Delimiting section start?
            if( tmp_delm == 0 ){                                // Delimiting section start occurrence
                tmp_delm = 1;                                   // Sets delimiting position.
                if( wkstr[i+1] != '"' ){
                    i++;
                    tmp_delmIndex = i;                          // Sets delimiting start position.
                }else{
                    tmp_delmIndex = i+1;
                }
            }else{                                              // Delimiting section end?
                memset( wkstr_wd, 0x00, sizeof( wkstr_wd ));
                if( i != tmp_delmIndex ){
                    memcpy( wkstr_wd, &wkstr[tmp_delmIndex],
                                    i - tmp_delmIndex );        // Sets delimiting position.
```

```
            }
            tmpStr = string( wkstr_wd );
            tmpCsvLineData.word.push_back( tmpStr );
            tmpCsvLineData.type.push_back( tmp_delm );
            tmp_delm = 0;
            // Sets next delimiting start position.
            i++;
            tmp_delmIndex = i;
            while(1){
               if( wkstr[i] == 0x00 ){
                  i--;
                  break;
               }
               if( delm.find( wkstr[i], 0 ) < delm.size() ){
                  i++;
                  tmp_delmIndex = i;
               }else{
                  i--;
                  break;
               }
            }
         }
      }else if( delm.find( wkstr[i], 0 ) < delm.size() ){
         if( tmp_delm == 1 ){                              // If character " has already appeared
            // the data is regarded as character data.
         }else{
            memset( wkstr_wd, 0x00, sizeof( wkstr_wd ));
            if( i != tmp_delmIndex ){
               memcpy( wkstr_wd, &wkstr[tmp_delmIndex],
                           i - tmp_delmIndex );          // Sets delimiting position.
            }
            tmpStr = string( wkstr_wd );
            tmpCsvLineData.word.push_back( tmpStr );
            tmpCsvLineData.type.push_back( tmp_delm );
            // Sets next delimiting start position.
            if( delm.find( wkstr[i+1], 0 ) < delm.size() ){       // If next is also delimiting position
               tmp_delmIndex = i+1;
               continue;
            }
            tmp_delmIndex = i;
            while(1){
               if( wkstr[i] == 0x00 ){
                  i--;
                  break;
               }
               if( delm.find( wkstr[i], 0 ) < delm.size() ){
                  i++;
                  tmp_delmIndex = i;
               }else{
                  i--;
                  break;
               }
            }
         }
      }
   }
   if( !(delm.find( wkstr[i-1], 0 ) < delm.size() ) ){
      memset( wkstr_wd, 0x00, sizeof( wkstr_wd ));
      memcpy( wkstr_wd, &wkstr[tmp_delmIndex],
                        i - tmp_delmIndex );                // Sets delimiting position.
      tmpStr = string( wkstr_wd );
      tmpCsvLineData.word.push_back( tmpStr );
      tmpCsvLineData.type.push_back( tmp_delm );
   }

   vCsvLineData.push_back( tmpCsvLineData );

   return true;
}
/**/
/*****************************************************************
 * Function name          : DeleteData
 * Function summary       : 1-line data deletion processing
 * Explanation            : 1-line data is deleted from memory.
 *                        :
 * Argument (input)       : None
 * Argument (output)      : None
```

```
 * Argument (I/O)          : None
 * Return value            : true : Normal    false : Failure
 * Created by              :
 * Updated on (created on) :
 * Remarks                 :
 ************************************************************************/
bool CCsvFile::DeleteData(void)
{
    if( p_vCsvLineData != vCsvLineData.end() ){
        if( vCsvLineData.empty() != true ){
            vCsvLineData.erase( vCsvLineData.begin(), vCsvLineData.end() );
            vCsvLineData.clear();
        }
    }else{
        return false;
    }
    p_vCsvLineData = vCsvLineData.end();

    return true;
}
/**/
/************************************************************************
 * Function name           : Strings
 * Function summary        : 1-line data acquisition processing
 * Explanation             : 1-line data is obtained from memory.
 *                         :
 * Argument (input)        : index : Record
 * Argument (output)       : None
 * Argument (I/O)          : None
 * Return value            : string  1-line data character string
 * Created by              :
 * Updated on (created on) :
 * Remarks                 :
 ************************************************************************/
string CCsvFile::Strings(int index)
{
    if( p_vCsvLineData != vCsvLineData.end() ){
        if(( index < (int)(p_vCsvLineData->word.size()) )&&
           ( index >= 0 )){
            return( p_vCsvLineData->word[index]);
        }
    }
    return("");
}
/**/
/************************************************************************
 * Function name           : GetDataCnt
 * Function summary        : 1-line data counter processing
 * Explanation             : Memory records are counted.
 *                         :
 * Argument (input)        : None
 * Argument (output)       : None
 * Argument (I/O)          : None
 * Return value            : int Record count
 * Created by              :
 * Updated on (created on) :
 * Remarks                 :
 ************************************************************************/
int  CCsvFile::GetDataCnt(void)
{
    if( p_vCsvLineData != vCsvLineData.end() ){
        return( (int)(p_vCsvLineData->word.size()) );
    }else{
        return( 0 );
    }
}

/**/
/************************************************************************
 * Function name           : TCalculateProc
 * Function summary        : Constructor
 * Explanation             : Class constructor
 *                         :
 * Argument (input)        : None
 * Argument (output)       : None
 * Argument (I/O)          : None
 * Return value            : None
```

```
 * Created by              :
 * Updated on (created on) :
 * Remarks                 :
 ************************************************************************/
 TCalculateProc::TCalculateProc()
{

    m_OutputData = "";                                          // Initializes output file name.

    return;
}
/**/
/************************************************************************
 * Function name           : ~  TCalculateProc
 * Function summary        : Destructor
 * Explanation             : Class destructor
 *                         :
 * Argument (input)        : None
 * Argument (output)       : None
 * Argument (I/O)          : None
 * Return value            : None
 * Created by              :
 * Updated on (created on) :
 * Remarks                 :
 ************************************************************************/
 TCalculateProc::~  TCalculateProc()
{
    // ******************************
    // Internal data clear
    // ******************************
    DataClear();                                                // Clears analysis data.

    return;
}
/**/
/************************************************************************
 * Function name           : Init
 * Function summary        : Analysis processing initialization
 * Explanation             : Convert processing is initialized.
 *                         :
 * Argument (input)        : None
 * Argument (output)       : None
 * Argument (I/O)          : None
 * Return value            : true : Normal    false : Failure
 * Created by              :
 * Updated on (created on) :
 * Remarks                 :
 ************************************************************************/
bool   TCalculateProc::Init()
{
    bool bRet;                                                  // Function return value

    // ******************************
    // Internal data clear
    // ******************************
    bRet = DataClear();                                         // Clears analysis data.
    if( bRet != true ){                                         // If return value contains error
        return( bRet );                                         // Process ends.
    }

    // ******************************
    // Environmental data read
    // ******************************
    bRet = EnvRead();                                           // Reads environmental data.
    if( bRet != true ){                                         // If return value contains error
        return( bRet );                                         // Process ends.
    }

    return(true);                                               // Returns if normal end.

}
/**/
/************************************************************************
 * Function name           : Init
 * Function summary        : Analysis processing initialization (with output file provided)
 * Explanation             : Convert processing is initialized (with output file provided).
 *                         :
```

```
 * Argument (input)       : FineName : Output file name
 * Argument (output)      : None
 * Argument (I/O)         : None
 * Return value           : true : Normal    false : Failure
 * Created by             :
 * Updated on (created on) :
 * Remarks                :
 ********************************************************************/
bool   TCalculateProc::Init(string FileName)
{
   bool bRet;

   m_OutputData = FileName;
   bRet = Init();

   return(bRet);                                                    // Returns if normal end.

}
/**/
/*********************************************************************
 * Function name          : DataClear
 * Function summary       : Processed data area clear
 * Explanation            : Processed data area is cleared.
 *                        :
 * Argument (input)       : None
 * Argument (output)      : None
 * Argument (I/O)         : None
 * Return value           : true : Normal    false : Failure
 * Created by             :
 * Updated on (created on) :
 * Remarks                :
 ********************************************************************/
bool   TCalculateProc::DataClear()
{

   // ****************************************
   // Check whether analysis data is stored,
   // and clear the data if stored.
   // ****************************************
   if( setCalculateData.empty() != true ){                          // If analysis data is stored
      setCalculateData.erase( setCalculateData.begin(),
                              setCalculateData.end() );             // Clears analysis data.
      setCalculateData.clear();
   }

   return( true );                                                  // Returns if normal.
}

/**/
/*********************************************************************
 * Function name          : EnvRead
 * Function summary       : Environmental data read processing
 * Explanation            : Environmental file is read and set in parameters.
 *                        :
 * Argument (input)       : None
 * Argument (output)      : None
 * Argument (I/O)         : None
 * Return value           : true : Normal    false : Failure
 * Created by             :
 * Updated on (created on) :
 * Remarks                :
 ********************************************************************/
bool   TCalculateProc::EnvRead()
{
   CCsvFile *EnvFileNames;
   CCsvFile *EnvDataName;
   double   fW1,fW2,fWMax;
   int      nRet, i;
   double   fKg ;
   double   fWeight;
   double   fGearHi;
   bool     bRet;
   string   szData;


   // ****************************************************************//
   // System parameters
```

```
// ****************************************************************//
// ----------------------
// Analysis interval (sec)
// ----------------------
m_fUnitTime = 1.0;                                          // Sets analysis interval in internal data.
// ----------------------
// Start gear position initial value
// ----------------------
m_nInitGear = 2;                                            // Sets starting gear position initial value in internal data.
// ----------------------
// Gear hold time (tg:sec)
// ----------------------
m_fTg = 3.0;                                                // Sets gear hold time (tg:sec) in internal data.
// ----------------------
// Select optimum gear position.
// ----------------------
m_fUd = 0.95;                                               // Sets final reduction ratio (transmission efficiency) to fixed value 0.95.
// ----------------------
// Inertial weight ratio equivalent in rotation section (E_FACT)
// ----------------------
m_fEFact = 0.03;                                            // Fixes E_FACT to 0.03.
// ----------------------
// Inertial weight ratio equivalent in rotation section (M_FACT)
// ----------------------
m_fMFact = 0.07;                                            // Fixes M_FACT to 0.07.
// -------------------------
// Weight per person
// -------------------------
m_fPersonW = 55.0;                                          // Weight per person (55kg)
// ****************************************************************//
// Clutch meet, clutch release
// ****************************************************************//
m_fClutch_Release = 4.0;                                    // Sets clutch release normalized engine speed in internal data.
m_fClutch_Meet = 5.0;                                       // Sets clutch meet normalized engine speed in internal data.
// ----------------------------
// Setting of circle circumference ratio to diameter
// ----------------------------
m_fPAI = 3.14;                                              // Sets circle circumference ratio to diameter in internal data.
m_fKg = 9.8;                                                // Sets gravitational acceleration.

// ---------------------------------------------------------
// Read environment definition file, and obtain environmental data.
// ---------------------------------------------------------
bRet = CommFun->FileExists(MAIN_ENVFILE);                   // Environment file exists?
if( bRet != true ){                                         // If non-existing
   return( false );
}

// ----------------------------
// Obtain environmental data file name.
// ----------------------------
EnvFileNames = new CCsvFile();
bRet = EnvFileNames->ReadFile(MAIN_ENVFILE);                // File read
if( bRet != true ){
   delete EnvFileNames;                                     // Deletes data read from environmental data file.
   return( false );
}

//-------------------------
// Pattern file exists
//-------------------------
EnvFileNames->SetAtRec(1);                                  // Moves to next record.
bRet = CommFun->FileExists(EnvFileNames->Strings(0));       // Pattern definition exists?
if( bRet == true ){                                         // If file exists
   nRet = PtnRead(EnvFileNames->Strings(0));                // Reads pattern file data.
   if( nRet == NG ){
      delete EnvFileNames;                                  // Deletes data read from environmental data file.
      return( false );
   }
}else{
   delete EnvFileNames;                                     // Deletes data read from environmental data file.
   return( false );
}

// -------------------------
// Environmental data read
// -------------------------
```

```
EnvFileNames->SetAtRec(2);
bRet = CommFun->FileExists(EnvFileNames->Strings(0));                    // If environmental data file definition exists
if( bRet != true ){                                                      // If non-existing
   delete EnvFileNames;                                                  // Deletes data read from environmental data file.
   return( false );
}
EnvDataName = new CCsvFile();
bRet = EnvDataName->ReadFile(EnvFileNames->Strings(0));                  // File read
if( bRet != true ){
   delete EnvDataName;                                                   // Deletes data read from environmental data file.
   delete EnvFileNames;                                                  // Deletes data read from environmental data file.
   return( false );
}

// -----------------------
// Curb vehicle weight
// -----------------------
EnvDataName->SetAtRec(1);                                                // Moves to first record.
szData = EnvDataName->Strings(0);                                        // Obtains curb vehicle weight.
if (szData != "" ){                                                      // If obtained data exists
   fW2 = atof(szData.c_str());                                           // Sets curb vehicle weight.
}else{                                                                   // If no obtained data exists
   delete EnvDataName;                                                   // Deletes data read from environmental data file.
   delete EnvFileNames;                                                  // Deletes data read from environmental data file.
   return( false );
}

// -----------------------
// Test payload
// -----------------------
EnvDataName->SetAtRec(2);                                                // Moves to second record.
szData = EnvDataName->Strings(0);                                        // Obtains max. payload.
if (szData != "" ){                                                      // If obtained data exists
   fWMax = atof(szData.c_str());                                         // Sets max. payload.
}else{                                                                   // If no obtained data exists
   delete EnvDataName;                                                   // Deletes data read from environmental data file.
   delete EnvFileNames;                                                  // Deletes data read from environmental data file.
   return( false );
}
// -----------------------
// Calculates half load.
// -----------------------
fW1 = fWMax /2.0;                                                        // Obtains test payload.

// -----------------------
// Riding capacity information
// -----------------------
EnvDataName->SetAtRec(3);                                                // Moves to third record.
szData = EnvDataName->Strings(0);                                        // Obtains test payload.
if (szData != "" ){                                                      // If obtained data exists
   m_fPersons = atof(szData.c_str());                                    // Obtains number of riding capacity.
}else{
   m_fPersons = 0.0;
}

// -----------------------
// Gravitational acceleration
// -----------------------
bRet = GetKG(fKg);                                                       // Obtains gravitational acceleration.
if (bRet == false){                                                      // If return value contains error
   delete EnvDataName;                                                   // Deletes data read from environmental data file.
   delete EnvFileNames;                                                  // Deletes data read from environmental data file.
   return( false );                                                      // Process ends.
}
fKg = fKg/1000.0;                                                        // kN
// -----------------------
// Setting of vehicle body weight and riding capacity weight data
// -----------------------
m_fCarMaxW = fWMax;                                                      // Sets max. payload (kg).
m_fCarlniW = fW2;                                                        // Sets empty vehicle mass (kg).
fWeight = m_fPersonW * m_fPersons;                                       // Sets riding capacity weight.
m_fCarMc   = fW1 ;                                                       // Test payload of car (kg)
m_fCarMe   = fW2 ;                                                       // Curb vehicle weight of car (kg)
m_fPersonM = fWeight ;                                                   // Weight of riding capacity (kg)

// -----------------------
// Overall vehicle height
```

```
// ------------------------
EnvDataName->SetAtRec(4);                                   // Moves to fourth record.
szData = EnvDataName->Strings(0);                           // Obtains overall height.
if (szData != "" ){                                     // If obtained data exists
    m_fOverHeight = atof(szData.c_str());                   // Obtains overall vehicle height.
}else{
    m_fOverHeight = 0.0;
}

// ------------------------
// Overall vehicle width
// ------------------------
EnvDataName->SetAtRec(5);                                   // Moves to fifth record.
szData = EnvDataName->Strings(0);                           // Obtains overall width.
if (szData != "" ){                                     // If obtained data exists
    m_fOverWidth = atof(szData.c_str());                    // Obtains overall vehicle width.
}else{
    m_fOverWidth = 0.0;
}

// ------------------------
// Tire dynamic rolling radius
// ------------------------
EnvDataName->SetAtRec(6);                                   // Moves to sixth record.
szData = EnvDataName->Strings(0);                           // Obtains tire radius.
if (szData != "" ){                                     // If obtained data exists
    m_fTarR = atof(szData.c_str());                         // Obtains tire rolling radius.
}else{
    m_fTarR = 0.0;
}

// ------------------------
// Top gear (Number of gear position)
// ------------------------
EnvDataName->SetAtRec(7);                                   // Moves to seventh record.
szData = EnvDataName->Strings(0);                           // Top gear read
if(szData == ""){                                       // If return value contains error
    m_nMaxGear = DEF_MAXGEAR;                               // Sets top gear to fixed value (7).
}else{                                                  // If read completes normally
    m_nMaxGear = atoi(szData.c_str());                      // Sets top gear in internal data.
}

// ------------------------
// Gear ratio read
// ------------------------
if( m_fGearHi.empty() != true ){                        // If gear ratio already exists
    m_fGearHi.erase( m_fGearHi.begin(), m_fGearHi.end() );      // Clears data.
    m_fGearHi.clear();
}
for( i = 1; i <= m_nMaxGear; i++ ){                     // Reads gear ratio.
    EnvDataName->SetAtRec(7+i);                             // Moves to "7+i"th record.
    szData = EnvDataName->Strings(0);                       // Reads gear ratio.
    if(szData == ""){                                   // If return value contains error
        fGearHi = 1.0;                                  // Sets gear ratio to 1.
    }else{                                              // If read completes normally
        fGearHi = atof(szData.c_str());                     // Sets gear ratio in internal data.
    }
    m_fGearHi.push_back( fGearHi );                         // Stores gear ratio.
}

// ------------------------
// Final reduction ratio
// ------------------------
EnvDataName->SetAtRec(8+m_nMaxGear);                        // Moves to "number of gear position + 8"th record.
szData = EnvDataName->Strings(0);                           // Acquisition of final reduction ratio.
if (szData != "" ){                                     // If obtained data exists
    m_fLastReduceGear = atof(szData.c_str());               // Obtains final reduction ratio.
}else{
    m_fLastReduceGear = 4.711;
}

// ------------------------
// Idling engine speed
// ------------------------
EnvDataName->SetAtRec(9+m_nMaxGear);                        // Moves to "number of gear position + 9"th record.
szData = EnvDataName->Strings(0);                           // Acquisition of idling engine speed
if (szData != "" ){                                     // If obtained data exists
```

```
    m_fIdleNe = atof(szData.c_str());                                   // Obtains idling engine speed.
  }else{
    m_fIdleNe = 500.0;
  }

  // ------------------------
  // Rated output engine speed
  // ------------------------
  EnvDataName->SetAtRec(10+m_nMaxGear);                                 // Moves to "number of gear position + 10"th record.
  szData = EnvDataName->Strings(0);                                     // Acquisition of rated output engine speed
  if (szData != "" ){                                                   // If obtained data exists
    m_fStandardOutputRotation = atof(szData.c_str());                   // Obtains rated output engine speed.
  }else{
    m_fStandardOutputRotation = 3000.0;
  }

  // ------------------------
  // Loaded limit engine speed
  // ------------------------
  EnvDataName->SetAtRec(11+m_nMaxGear);                                 // Moves to "number of gear position + 11"th record.
  szData = EnvDataName->Strings(0);                                     // Acquisition of loaded limit engine speed
  if (szData != "" ){                                                   // If obtained data exists
    m_fOutputRotation = atof(szData.c_str());                           // Obtains loaded limit engine speed.
  }else{
    m_fOutputRotation = 3100.0;
  }
  // ------------------------
  // Rated engine speed
  // ------------------------
  m_fFixedNe = GetMaxNe();                                              // Rated engine speed setting

  delete EnvDataName;                                                   // Deletes data read from environmental data file.

  // ************************************************//
  // Setting of clutch meet and release revolutions //
  // ************************************************//
  m_fClutch_ReleaseNe = ( m_fFixedNe - m_fIdleNe ) *
                        m_fClutch_Release/100.0 + m_fIdleNe;            // Calculates clutch release engine speed.
  m_fClutch_MeetNe    = ( m_fFixedNe - m_fIdleNe ) *
                        m_fClutch_Meet/100.0 + m_fIdleNe;               // Calculates clutch meet engine speed.

  // ------------------------
  // Max. torque data
  // ------------------------
  EnvFileNames->SetAtRec(3);
  bRet = CommFun->FileExists(EnvFileNames->Strings(0));                 // Torque file definition exists?
  if( bRet != true ){                                                   // If no file exists
    delete EnvFileNames;                                                // Deletes data read from environmental data file.
    return( false );
  }
  nRet = ReadMaxTorqueData(EnvFileNames->Strings(0));                   // Acquisition of max. torque data
  if (nRet == NG){                                                      // If return value contains error
    delete EnvFileNames;                                                // Deletes data read from environmental data file.
    return( false );                                                    // Process ends.
  }

  delete EnvFileNames;                                                  // Deletes data read from environmental data file.

  // ****************************************//
  // Setting of excess force ratio data
  // ****************************************//
  nRet = GetExceedF();                                                  // Obtains excess force ratio.
  if (nRet == NG){                                                      // If return value contains error
    return( false );                                                    // Process ends.
  }

  return( true );                                                       // Returns if normal.
}
/**/
/*********************************************************************
 * Function name          : CalculateProcess
 * Function summary       : Main processing of Convert
 * Explanation            : Main processing of Convert is executed.
 *                        :
 * Argument (input)       : None
 * Argument (output)      : None
 * Argument (I/O)         : None
```

```
 * Return value            : OK : Normal   NG: Failure
 * Created by              :
 * Updated on (created on) :
 * Remarks                 :
 ********************************************************************/
int   TCalculateProc::CalculateProcess()
{
    bool bRet;

    bRet = Calculate_progress1();                                   // Calculates reference vehicle speed and acceleration.
    if( bRet == false ){
        return( NG );
    }

    bRet = Calculate_progress2();                                   // Makes compatible with previous version.
    if( bRet == false ){
        return( NG );
    }

    bRet = Calculate_T1T2Set();                                     // Calculates T1 and T2.
    if( bRet == false ){
        return( NG );
    }

    bRet = Calculate_progress3();                                   // Determines gear position, and sets parameters.

    if( bRet == false ){
        return( NG );
    }

    DispCalculateData();                                            // Displays parameter information.
    // Data write
    WriteAllCalculateData();

    return( OK );
}
/**/
/*********************************************************************
 * Function name           : PtnRead
 * Function summary        : Pattern setup processing
 * Explanation             : Pattern data for Convert processing is set.
 *                         :
 * Argument (input)        : szFile : read pattern file name
 * Argument (output)       : None
 * Argument (I/O)          : None
 * Return value            : true : Normal    false : Failure
 * Created by              :
 * Updated on (created on) :
 * Remarks                 :
 ********************************************************************/
bool    TCalculateProc::PtnRead(string szFile)
{
    map<double,stCalculateData>::iterator p_setStart;
    map<double,stCalculateData>::iterator p_setEnd;
    map<double,stCalculateData>::iterator p_next;                   // Next pointer
    stCalculateData tmpCalculateData;
    string  szData;
    string  szPtn1015File;
    string  szTmp;
    FILE    *fp;
    int     row;
    char    *p, buf[200];
    double  tmpTime;                                                // Obtained time
    double  tmpAllTime;                                             // Accumulated time
    double  tmpSetTime;                                             // Set time
    double  tmpBefTime;                                             // Previous set time
    double  tmpfV;                                                  // Vehicle speed
    int     nGear;                                                  // Selected gear
    double  p1_data;
    double  p2_data;

    nGear = 0;
    p1_data = 0.0;
    p2_data = 0.0;
    tmpAllTime = 0.0;                                               // Initializes accumulated seconds
    //-----------------------------------------
    // Pattern file read processing Step↓7
```

```
// (Obtain accumulated time.)
//-----------------------------------------
if((fp = fopen( szFile.c_str(), "rt" ) ) == NULL ){                    // If file open failed
   sprintf( buf, "%s\n\nThe name file is not found.",
                                    szFile.c_str() );                  // Sets error message.
   cout << buf << endl;
   return( false );                                                    // Ends with error.
}else{                                                                 // If normally opened
   for( row=0; fgets( buf, BFSZ, fp );row++ ){
      if( row == 0 ) continue;
      p = strtok( buf, " \n\t" );
      if( p == NULL ) continue;                                        // If data has no vehicle speed and time
      tmpTime = atof( p );
      p = strtok( NULL, "\n\t" );
      if( p == NULL ) continue;                                        // If data has no vehicle speed and time
      tmpAllTime = tmpTime;                                            // Sets accumulated seconds.
   }
   fclose(fp);
}

//-----------------------------------------
// If current pattern file exists
//-----------------------------------------
if( setCalculateData.empty() != true ){                                // If analysis data exists
   setCalculateData.erase( setCalculateData.begin(),
                           setCalculateData.end() );                   // Deletes existing analysis data
   setCalculateData.clear();                                           // _____
}

//-----------------------------------------
// Create pattern file null data.
//-----------------------------------------
memset( &tmpCalculateData, 0x00, sizeof( tmpCalculateData ));          // Initializes temporary analysis data
for( tmpSetTime = 0.0; tmpSetTime < tmpAllTime;
                  tmpSetTime = tmpSetTime + m_fUnitTime ){             // Sets data for analysis interval according to accumulated time.
   tmpCalculateData.fTimes = tmpSetTime *10;                           // Sets accumulated time in msec.
   tmpCalculateData.nWriteFlg = 1;                                     // Analysis write ON
   setCalculateData.insert(pair<double,stCalculateData>
            (tmpCalculateData.fTimes, tmpCalculateData) );             // Sets analysis time in pattern information
}

//-----------------------------------------
// Pattern file read processing Step1
// (Set in analysis data file)
//-----------------------------------------
if((fp = fopen( szFile.c_str(), "rt" ) ) == NULL ){                    // If file open failed
   sprintf( buf, "%s\n\nThe name file is not found.",
                                    szFile.c_str() );                  // Sets error message.
   cout << buf << endl;
   return( false );                                                    // Ends with error.
}else{                                                                 // If normally opened
   tmpAllTime = 0.0;                                                   // Initializes accumulated seconds.
   tmpBefTime = 0.0;
   for( row=0; fgets( buf, BFSZ, fp ); row++){
      if( row == 0 ) continue;
      p = strtok( buf, " \n\t" );
      if( p != NULL ){
         p1_data = atof( p );
      }
      p = strtok( NULL, "\n\t" );
      if( p == NULL ) continue;                                        // If data has no vehicle speed and time
      if( strcmp( p, "IDLE" ) == 0 ){
         p2_data = 0.0;
      }else{
         p2_data = atof( p );
      }
      memset( &tmpCalculateData, 0x00, sizeof( tmpCalculateData ));    // Initializes temporary analysis data
      //-------------------------
      // Set vehicle speed in internal data.
      //-------------------------
      tmpCalculateData.nPtnReadFlg = 1;                                // Sets pattern read flag to ON.
      if( p2_data == 0 ){                                              // IDLE ?
         tmpfV = 0.0;                                                  // Sets vehicle speed to 0.
         tmpCalculateData.bIdle  = true;                               // IDLE state ON
      }else{
         tmpfV = p2_data;
         //---- Caution-----------------------------------------
```

```
            // Clutch ON/OFF may be changed during subsequent analysis.
            //-------------------------------------------------
            tmpCalculateData.bIdle  = false;                                    // IDLE state OFF
        }
        tmpCalculateData.fV = tmpfV;                                            // Sets speed.

        //-------------------------------------
        // Set time and accumulated seconds in internal data.
        //-------------------------------------
        tmpTime = p1_data;
        tmpCalculateData.fT = tmpTime - tmpBefTime;                             // Sets number of seconds
        tmpCalculateData.fTimes = tmpTime *10;                                  // Sets accumulated seconds in msec.

        //-------------------------
        // Gear description check
        //-------------------------
        p = strtok( NULL, " \n\t" );
        if( p != NULL ){                                                        // If gear is described
            if( *p != 'N' ){                                                    // Neutral?
                nGear = atoi( p );
                tmpCalculateData.nGear = nGear;                                 // Sets gear.
            }else{
                tmpCalculateData.nGear = 0;                                     // Sets gear.
                nGear = 0;                                                      // ----------
            }
        }else{                                                                  // In case of normal pattern file
            tmpCalculateData.nGear = nGear;                                     // Sets gear.
        }

        //-----------------------------------------------
        // Set pattern file data as analysis data
        //-----------------------------------------------
        p_setCalculateData = setCalculateData.find( tmpCalculateData.fTimes );      // Searches for target data based on accumulated seconds
        if( p_setCalculateData != setCalculateData.end() ){                     // If search succeeds
            tmpCalculateData.nWriteFlg = 1;                                     // Analysis setting
            setCalculateData.erase( p_setCalculateData );                       // Deletes data once
            setCalculateData.insert(pair<double,stCalculateData>
                    (tmpCalculateData.fTimes, tmpCalculateData) );              // Sets read data
        }else{
            tmpCalculateData.nWriteFlg = 1;                                     // Analysis setting
            setCalculateData.insert(pair<double,stCalculateData>
                    (tmpCalculateData.fTimes, tmpCalculateData) );              // Sets read data
        }

        //-------------------------------------------------
        // Fill with data up to next gear in case of gear description mode.
        //-------------------------------------------------
        if( tmpTime - tmpBefTime != m_fUnitTime ){                              // In case of different time interval and in gear description mode
            p_setEnd = setCalculateData.find( tmpCalculateData.fTimes );            // Searches for target data based on the accumulated seconds.
            tmpCalculateData.fTimes = tmpTime *10;                              // Sets number of seconds.
            p_setStart = setCalculateData.find( tmpBefTime );                   // Searches for target data based on accumulated seconds.
            for( p_setCalculateData = p_setStart;
                 p_setCalculateData != p_setEnd; p_setCalculateData++ ){
                p_setCalculateData->second.nGear = nGear;                       // Sets gear
            }
        }
        tmpAllTime = tmpTime;                                                   // Accumulated seconds
        tmpBefTime = tmpTime;
    }
    fclose(fp);
  }

  BtwnTimeSet( setCalculateData.begin(), setCalculateData.end() );              // Updates all section time

  return( true );
}
/**/
/*****************************************************************
 * Function name          : GetMaxNe
 * Function summary       : Max. engine speed acquisition processing
 * Explanation            : Rated output engine speed is obtained and set.
 *                        :
 * Argument (input)       : None
 * Argument (output)      : None
 * Argument (I/O)         : None
 * Return value           : double  Rated output engine speed
 * Created by             :
```

```
 * Updated on (created on) :
 * Remarks                 :
 ************************************************************************/
double   TCalculateProc::GetMaxNe()
{
   double fMaxNe;                                                  // Max. engine speed

   // Rated output engine speed
   fMaxNe = (int)m_fStandardOutputRotation;                        // Sets rated output engine speed.
   return( fMaxNe );
}
/**/
/************************************************************************
 * Function name          : GetKG
 * Function summary       : Gravitational acceleration acquisition processing
 * Explanation            : Gravitational acceleration is obtained and set.
 *                        :
 * Argument (input)       : None
 * Argument (output)      : None
 * Argument (I/O)         : fKg : Gravitational acceleration
 * Return value           : true : Normal    false : Failure
 * Created by             :
 * Updated on (created on) :
 * Remarks                :
 ************************************************************************/
bool   TCalculateProc::GetKG(double &fKg)
{

   fKg = m_fKg;                                                    // Sets gravitational acceleration.
   return(true);

}
/**/
/************************************************************************
 * Function name          : GetExceedF
 * Function summary       : Data setup processing for excess force ratio
 * Explanation            : Force ratio data is set.
 *                        :
 * Argument (input)       : None
 * Argument (output)      : None
 * Argument (I/O)         : None
 * Return value           : true : Normal    false : Failure
 * Created by             :
 * Updated on (created on) :
 * Remarks                :
 ************************************************************************/
bool   TCalculateProc::GetExceedF(void)
{
   stExceedForce tmpExceedForce;                                   // Temporary excess force ratio data
   int      nRet;                                                  // Function return value
   int      i;
   string   tmpStr;
   double   fGearti;
   string   szKey, szData;

   // Existing excess force ratio data is provided.
   if( m_ExceedForce.empty() != true ){                            // If data already exists
      m_ExceedForce.erase( m_ExceedForce.begin(),
                  m_ExceedForce.end() );                           // Deletes existing data.
      m_ExceedForce.clear();
   }

   for( i = 1; i <= m_nMaxGear; i++ ){
      memset( &tmpExceedForce, 0x00, sizeof( tmpExceedForce ));    // Initializes temporary data.

      tmpExceedForce.nGear = i;                                    // Gear position

      // Gear position
      nRet = GetGearHi(tmpExceedForce.nGear,fGearti);
      if( nRet != OK ){
         return( false );
      }
      tmpExceedForce.fGearti = fGearti;                            // Gear ratio
      // Transmission efficiency
      tmpExceedForce.fForcePer = GetGearPass( i );                 // Sets gear transmission efficiency.

      // Set excess torque ratio and normalized engine speed based on gear value.
```

```
        if( i <= 2 ){                                           // If gear position is 2-speed or less
            if( m_fCarMaxW + m_fCarIniW + (m_fPersonW * m_fPersons) >=
                                    DEF_FORCEOVER ){            // GVW (empty vehicle mass + max. payload) of 8t or more
                tmpExceedForce.fFreePer = DEF_FORCE_OV_GEAR2;           // Excess torque ratio 2.0
            }else{
                tmpExceedForce.fFreePer = DEF_FORCE_UN_GEAR2;           // Excess torque ratio 2.4
            }
            tmpExceedForce.fMinPer  = DEF_FORCE_NE_GEAR2;               // Lower-limit engine speed (normalized engine speed) 5%
        }else if( i == 3 ){                                     // If gear position is 3-speed
            if( m_fCarMaxW + m_fCarIniW + (m_fPersonW * m_fPersons) >=
                                    DEF_FORCEOVER ){            // GVW (empty vehicle mass + max. payload) of 8t or more
                tmpExceedForce.fFreePer = DEF_FORCE_OV_GEAR3;           // Excess torque ratio 1.7
            }else{
                tmpExceedForce.fFreePer = DEF_FORCE_UN_GEAR3;           // Excess torque ratio 1.7
            }
            tmpExceedForce.fMinPer  = DEF_FORCE_NE_GEAR3;               // Lower-limit engine speed (normalized engine speed) 11%
        }else if( i == 4 ){                                     // If gear position is 4-speed
            if( m_fCarMaxW + m_fCarIniW + (m_fPersonW * m_fPersons) >=
                                    DEF_FORCEOVER ){            // GVW (empty vehicle mass + max. payload) of 8t or more
                tmpExceedForce.fFreePer = DEF_FORCE_OV_GEAR4;           // Excess torque ratio 1.3
            }else{
                tmpExceedForce.fFreePer = DEF_FORCE_UN_GEAR4;           // Excess torque ratio 1.6
            }
            tmpExceedForce.fMinPer  = DEF_FORCE_NE_GEAR4;               // Lower-limit engine speed (normalized engine speed) 19%
        }else{                                                  // If gear position is 5-speed or more
            if( m_fCarMaxW + m_fCarIniW + (m_fPersonW * m_fPersons) >=
                                    DEF_FORCEOVER ){            // GVW (empty vehicle mass + max. payload) of 8t or more
                tmpExceedForce.fFreePer = DEF_FORCE_OV_GEAR4;           // Excess torque ratio 1.3
            }else{
                tmpExceedForce.fFreePer = DEF_FORCE_UN_GEAR4;           // Excess torque ratio 1.6
            }
            tmpExceedForce.fMinPer  = DEF_FORCE_NE_GEAR5;               // Lower-limit engine speed (normalized engine speed) 26%
        }

        tmpExceedForce.fMinNe = ( m_fFixedNe - m_fIdleNe ) *
                        tmpExceedForce.fMinPer/100.0 + m_fIdleNe;       // Calculates lower-limit engine speed.
        m_ExceedForce.insert( tmpExceedForce );                         // Sets excess force ratio data.
    }

    return( true );
}
/**/
/*****************************************************************
 * Function name          : GetGearPass
 * Function summary       : Gear transmission efficiency setup processing
 * Explanation            : Gear transmission efficiency is set.
 *                        :
 * Argument (input)       : nGear : Gear position
 * Argument (output)      : None
 * Argument (I/O)         : None
 * Return value           : double Transmission efficiency
 * Created by             :
 * Updated on (created on) :
 * Remarks                :
 *****************************************************************/
double TCalculateProc::GetGearPass( int nGear )
{

    if( m_fGearHi.empty() == true ){
        return( 0 );
    }
    if( nGear > (int)(m_fGearHi.size()) ){
        return( 1 );
    }
    // ----------------------------
    // Set transmission efficiency based on gear ratio.
    // ----------------------------
    if( m_fGearHi[nGear-1] == 1 ){                              // If gear ratio is 1:0.
        return( DEF_FORCE_ON98 );
    }else{
        return( DEF_FORCE_OFF95 );
    }
}
/**/
/*****************************************************************
 * Function name          : ReadMaxTorqueData
 * Function summary       : Max. torque data setup processing
```

```
 * Explanation            : Data is read from max. torque data file.
 *                        :
 * Argument (input)       : FileName : Max. torque data file name
 * Argument (output)      : None
 * Argument (I/O)         : None
 * Return value           : OK : Normal   NG : Failure
 * Created by             :
 * Updated on (created on) :
 * Remarks                :
 ************************************************************************/
int   TCalculateProc::ReadMaxTorqueData(string FileName)                    // Obtains max. torque data.
{
    MAX_TORQUE  tmpMaxTorque;                                               // Temporary max. torque data
    long        row;
    string      szTmp;
    double      fData;
    char        buf[100];
    FILE        *fp;
    CCsvFile    *pStringList;                                               // Template

    // ----------------------
    // Rated torque
    // ----------------------
    m_fStandardTorque = 0;                                                  // Sets rated torque data in internal data.

    // ---------------------------------------
    // Max. torque data file open processing
    // ---------------------------------------
    if( ( fp = fopen( FileName.c_str(), "rt" ) )
                                        == NULL ){                          // If no applicable file exists
        sprintf( buf, "%s\n\nThe file is not found.",
                FileName.c_str() );                                         // Generates dialog message.
        cout << buf << endl;
        return( NG );                                                       // Ends with error
    }

    // --------------------------------
    // Deletes existing data if any.
    // --------------------------------
    if (m_MaxTorque.empty() != true){                                       // If data exists
        m_MaxTorque.erase( m_MaxTorque.begin(),
                           m_MaxTorque.end() );                             // Deletes existing data.
        m_MaxTorque.clear();                                                // _________
    }

    pStringList   = new CCsvFile();                                         // Generates template.
    // ------------------------------------------------
    // Read file and store internal data.
    // ------------------------------------------------
    for( row = 0; fgets( buf, sizeof(buf), fp ); row ++){
        // ---------------------------
        // Skip comment section.
        // ---------------------------
        if (row < 1 ) continue;                                             // Does not process comment section.

        // -------------------------------------------
        // Process read data based on tab delimiting.
        // -------------------------------------------
        pStringList->DeleteData();
        pStringList->SetDataStr( string(buf) );                             // Stores read data.
        pStringList->SetAtRec(1);
        // ---------------------------
        // Max. torque data exists?
        // ---------------------------
        if( pStringList->GetDataCnt() == 2 ){                               // If column data count is 2
            CommFun->AStrToDouble(pStringList->Strings(0), fData);          // Converts character string to real number.
            tmpMaxTorque.fEgrevo = fData;                                   // Sets engine speed.

            CommFun->AStrToDouble(pStringList->Strings(1), fData);          // Converts character string to real number.
            tmpMaxTorque.fEgtq = fData;                                     // Sets engine torque.
            // ---------------------------------------
            // Store max. torque data in internal area.
            // ---------------------------------------
            m_MaxTorque.insert( tmpMaxTorque );                             // Stores max. torque data in internal data.
            // ------------------
            // Rated torque determination
            // ------------------
```

```
            if( fData > m_fStandardTorque ){                    // Max. torque data?
                m_fStandardTorque = fData;                      // Sets rated torque data in internal data.
            }
        }
    }
    delete pStringList;                                         // Deletes template.

    fclose(fp);
    return( OK );
}
/**/
/*****************************************************************
 * Function name          : GetLineReviseMaxTorque
 * Function summary       : Max. torque data interpolation processing
 * Explanation            : Obtain max. torque data, and execute calculation. (Linear interpolation)
 *                        :
 * Argument (input)       : fNe : Engine speed
 * Argument (output)      : None
 * Argument (I/O)         : None
 * Return value           : double Torque
 * Created by             :
 * Updated on (created on) :
 * Remarks                :
 *****************************************************************/
double   TCalculateProc::GetLineReviseMaxTorque(double fNe)
{
    double  fNeA,fNeB,fTorqueA,fTorqueB,fMaxTorque;
    set<MAX_TORQUE>::iterator p_nextMaxTorque;                  // Pointer
    MAX_TORQUE  tmpMaxTorque;                                   // Temporary max. torque data


    // Check max. torque data within data range if it exists.
    if (m_MaxTorque.empty() != true ){
    }else{
        // Return NG if max. torque data does not exist.
        return( 0.0 );
    }

    // Find appropriate engine speed and max. loss torque data from array.
    memset( &tmpMaxTorque, 0x00, sizeof( tmpMaxTorque ));
    tmpMaxTorque.fEgrevo = fNe;
    p_MaxTorque = m_MaxTorque.lower_bound( tmpMaxTorque );

    // Return NG if not found.
    if( p_MaxTorque == m_MaxTorque.end() ){
        p_MaxTorque = m_MaxTorque.end();
        p_MaxTorque--;
        fNeB = p_MaxTorque->fEgrevo;
        fTorqueB = p_MaxTorque->fEgtq;
        p_MaxTorque--;
        fTorqueA = p_MaxTorque->fEgtq;
        fNeA = p_MaxTorque->fEgrevo;
        // Prevent dividing by 0.
        if ((fNeB - fNeA) == 0) return( 0.0 );
        // Obtain appropriate max. torque by linear interpolation (polygonal line).
        fMaxTorque = fTorqueB +
            (fTorqueB - fTorqueA) /
            (fNeB - fNeA) *
            (fNe - fNeB);
        return( fMaxTorque );
    }

    fNeB = p_MaxTorque->fEgrevo;
    fTorqueB = p_MaxTorque->fEgtq;
    // Obtain preceding data.
    if( p_MaxTorque != m_MaxTorque.begin() ){
        p_MaxTorque--;
        fTorqueA = p_MaxTorque->fEgtq;
        fNeA = p_MaxTorque->fEgrevo;
    }else{                                                      // Next data if first data.
        fTorqueA = p_MaxTorque->fEgtq;
        fNeA = p_MaxTorque->fEgrevo;
        p_MaxTorque++;
        fNeB = p_MaxTorque->fEgrevo;
        fTorqueB = p_MaxTorque->fEgtq;
    }
```

```
    // Prevent dividing by 0.
    if ((fNeB - fNeA) == 0) return( 0.0 );

    // Obtain appropriate max. torque by linear interpolation (polygonal line).
    fMaxTorque = fTorqueA +
          (fTorqueB - fTorqueA) /
          (fNeB - fNeA) *
          (fNe - fNeA);

    return( fMaxTorque );
}
/**/
/*****************************************************************
 * Function name          : CalcForceWithNeSet
 * Function summary       : Driving force setup confirmation processing
 * Explanation            : Driving force is calculated. Also, shift-up availability is determined.
 *                        :
 * Argument (input)       : nGear : Gear position to be used
 * Argument (input)       : fTe : Required rotation torque
 * Argument (input)       : fNe : Engine speed
 * Argument (output)      : fF  : Required force ratio
 * Argument (I/O)         : None
 * Return value           : true : shift up OK       false : shift up NG
 * Created by             :
 * Updated on (created on) :
 * Remarks                :
 *****************************************************************/
bool   TCalculateProc::CalcForceWithNeSet(int nGear, double fTe,
                           double fNe, double &fF )
{
    stExceedForce tmpExceedForce;                                   // Data for excess force ratio
    double fGtn;                                                    // Speed change gear ratio
    double fFper;                                                   // Power transmission efficiency
    double fMaxTe;                                                  // Max. torque
    bool   bRet;                                                    // Max. force ratio/required force ratio

    if( nGear != 0 ){                                               // If gear position is set
        memset( &tmpExceedForce, 0x00, sizeof( tmpExceedForce ));   // Initializes search data.
        tmpExceedForce.nGear = nGear;                               // Sets gear position for search target.
        p_ExceedForce = m_ExceedForce.find( tmpExceedForce );       // Obtains applicable gear position.
        fGtn = p_ExceedForce->fGearti;                              // Obtains gear ratio.
        fFper = p_ExceedForce->fForcePer;                           // Sets power transmission efficiency.

        if(( fTe != 0.0 )&&( fGtn != 0.0 )&&( fFper != 0.0 )){
            fF = fTe * fGtn * fFper / ( 1000.0 * m_fTarR );         // Sets driving force.
        }
    }

    // Obtain max. torque.
    fMaxTe = GetLineReviseMaxTorque(fNe);
    if( p_ExceedForce->fFreePer < fMaxTe / fTe ){
        bRet = true;                                                // shift up OK.
    }else{
        bRet = false;                                               // shift up NG.
    }

    return(bRet);
}
/**/
/*****************************************************************
 * Function name          : CheckForce
 * Function summary       : Driving force calculation processing
 * Explanation            : Driving force is calculated. Shift-up availability is determined.
 *                        :
 * Argument (input)       : nGear : Gear position to be used
 * Argument (input)       : fVana : Analysis speed
 * Argument (input)       : fCarA : Acceleration
 * Argument (output)      : None
 * Argument (I/O)         : None
 * Return value           : true : shift up OK       false : shift up NG
 * Created by             :
 * Updated on (created on) :
 * Remarks                :
 *****************************************************************/
bool   TCalculateProc::CheckForce(int nGear, double fVana, double fCarA )
{
    double fTe;                                                     // Required rotation torque
```

```
    double fNe;                                                 // Number of engine speed
    double fF;                                                  // Driving force
    int    nRet;                                                // Function return value
    bool   bRet;                                                // Function return value

    nRet = GetNe( nGear, fVana, fNe);                                   // Engine speed
    if( nRet == NG ){
        return( false );
    }
    nRet = GetTe( nGear, fVana, fCarA, fNe, fTe);                       // Engine torque
    if( nRet == NG ){
        return( false );
    }

    bRet = CalcForceWithNeSet(nGear, fTe, fNe, fF );                    // Determines shift-up availability based on torque.
    return( bRet );

}
/**/
/*****************************************************************
 * Function name          : CalcTeMaxSp
 * Function summary       : Target-speed follow calculation processing
 * Explanation            : When speed changes from A to B,
 *                        : speed fV is calculated if target-speed follow is impossible in time fTm.
 * Argument (input)       : nGear : Gear position to be used
 * Argument (input)       : fVbef : Previous speed
 * Argument (input)       : fTm   : Usage time
 * Argument (output)      : fNe   : engine speed
 * Argument (I/O)         : double fV : Speed for this time/speed after re-calculation
 * Return value           : true : Converged   false : Not converged
 * Created by             :
 * Updated on (created on) :
 * Remarks                :
 *****************************************************************/
bool   TCalculateProc::CalcTeMaxSp(int nGear, double fTm, double fVbef, double &fV, double &fNe )
{
    double fV_def;                                              // Vehicle speed before change
    double fV1;                                                 // Calculation base point speed 1
    double fV2;                                                 // Calculation base point speed 2
    double fV_cal1;                                             // Calculation point intermediate-1
    double fV_calM;                                             // Calculation point intermediate
    double fV_cal2;                                             // Calculation point intermediate+1
    double fTe_DIF_1;                                           // Torque ratio of calculation point intermediate-1
    double fTe_DIF_M;                                           // Torque ratio of calculation point intermediate
    double fTe_DIF_2;                                           // Torque ratio of calculation point intermediate+1
    double fNe_def;                                             // Engine speed for this time (no correction)
    double fNe_cal;                                             // Engine speed for this time (after calculation)
    double fTe_def;                                             // Torque for this time (no correction)
    double fTe_spl;                                             // Torque for this time (corrected)
    double fTe_cal;                                             // Torque for this time (after calculation)
    double fCarA;                                               // Acceleration
    int    nRet;                                                // Function return value
    int    i;

    fV_def = fV;                                                // Stores vehicle speed before setting.

    // Calculate acceleration, torque, and engine speed for this time.
    nRet = GetNe( nGear, fV, fNe_def);                                  // Engine speed
    if( nRet == NG ){
        return( false );
    }

    if( fNe_def >= m_fOutputRotation ){
        fNe_def = m_fOutputRotation;
        nRet = GetV( nGear, fNe_def, fV );
        if( nRet == NG ){
            return( false );
        }
        fV_def = fV;
    }

    fNe = fNe_def;

    fCarA = (( fV - fVbef) / fTm) * 1000.0/(60.0*60.0);                 // Calculates acceleration.

    if( fCarA <= 0 ){
        return( true );
```

```
}
nRet = GetTe_NotRevise( nGear, fV, fCarA, fNe_def, fTe_def);                    // Engine torque (without complement)
if( nRet == NG ){
   return( false );
}
fTe_spl = GetLineReviseMaxTorque(fNe_def);

if( fTe_spl < fTe_def ){                                                        // If spline-complemented torque is
   nRet = GetNe( nGear, fV, fNe_def);                                           // Engine speed
   if( nRet == NG ){
      return( false );
   }
   if( fNe_def < m_fClutch_MeetNe ){
      fNe_def = m_fClutch_MeetNe;
   }
   nRet = GetTe_NotRevise( nGear, fV, fCarA, fNe_def, fTe_def);                 // Engine torque (without complement)
   if( nRet == NG ){
      return( false );
   }
                                                                    // smaller than calculated torque
   fTe_cal = fTe_def;                                                           // Torque to be calculated
   fNe_cal = fNe_def;                                                           // Engine speed to be calculated
   fV1 = fVbef;
   fV2 = fV_def;
   for( i=0; i<10000; i++ ){                                                    // Finds approximate TeMax=Te and acceleration.
      if( fTe_spl != 0 ){
         if(( 0 <= ( fTe_spl - fTe_cal) )&&
            ( ( fTe_spl - fTe_cal  ) < 1.0E-6 )){                               // If nearest value is found
            break;
         }
      }

      // ---------------------------------
      // Calculate intermediate speed.
      // ---------------------------------
      fV = fV1 + (fV2 - fV1)/2.0;                                               // Determines mid-point speed.
      fV_calM = fV;
      // Calculate intermediate acceleration and torque (with/without correction).
      fCarA = (( fV - fVbef) / fTm) * 1000.0/(60.0*60.0);                       // Calculates acceleration.
      // Calculate mid-point engine speed.
      nRet = GetNe( nGear, fV, fNe_cal);                                        // Engine speed
      if( nRet == NG ){
         return( false );
      }
      if( fNe_cal < m_fClutch_MeetNe ){
         fNe_cal = m_fClutch_MeetNe;
      }
      if( fNe_cal >= m_fOutputRotation ){
         fNe_cal = m_fOutputRotation;
         nRet = GetV( nGear, fNe_cal, fV );
         if( nRet == NG ){
            return( false );
         }
         fV_calM = fV;
         fCarA = (( fV - fVbef) / fTm) * 1000.0/(60.0*60.0);
      }
      fNe = fNe_cal;
      nRet = GetTe_NotRevise( nGear, fV, fCarA, fNe_cal, fTe_cal);              // Engine torque (without complement)
      if( nRet == NG ){
         return( false );
      }
      fTe_spl = GetLineReviseMaxTorque(fNe_cal);                                // Linear interpolation

      if( fTe_spl != 0 ){
         if(( 0 <= ( fTe_spl - fTe_cal) )&&
            ( ( fTe_spl - fTe_cal  ) < 1.0E-6 )){                               // If nearest value is found
            break;
         }
         fTe_DIF_M = (fTe_cal / fTe_spl );
      }else{
         fTe_DIF_M = 0;
      }

      // ---------------------------------------------
      // Calculate speed faster than mid-point speed.
      // ---------------------------------------------
      fV = fV1 + (fV2 - fV1)/2.0 + (fV2 - fV1)/ 4.0;
```

```
fV_cal2 = fV;
// Calculate acceleration and torque (with/without correction).
fCarA = (( fV - fVbef) / fTm )* 1000.0/(60.0*60.0);                    // Calculates acceleration.

// Calculate engine speed.
nRet = GetNe( nGear, fV, fNe_cal);                                    // Engine speed
if( nRet == NG ){
   return( false );
}
if( fNe_cal < m_fClutch_MeetNe ){
   fNe_cal = m_fClutch_MeetNe;
}
if( fNe_cal >= m_fOutputRotation ){
   fNe_cal = m_fOutputRotation;
   nRet = GetV( nGear, fNe_cal, fV );
   if( nRet == NG ){
      return( false );
   }
   fV_cal2 = fV;
   fCarA = (( fV - fVbef) / fTm) * 1000.0/(60.0*60.0);
}
fNe = fNe_cal;
nRet = GetTe_NotRevise( nGear, fV, fCarA, fNe_cal, fTe_cal);           // Engine torque (without complement)
if( nRet == NG ){
   return( false );
}
fTe_spl = GetLineReviseMaxTorque(fNe_cal);                            // Linear interpolation

if( fTe_spl != 0 ){
   if(( 0 <= ( fTe_spl - fTe_cal) )&&
      ( ( fTe_spl - fTe_cal  ) < 1.0E-6 )){                           // If nearest value is found
       break;
   }
   fTe_DIF_2 = (fTe_cal / fTe_spl );
}else{
   fTe_DIF_2 = 0;
}

// ---------------------------------------------
// Calculate speed slower than mid-point speed.
// ---------------------------------------------
fV = fV1 + (fV2 - fV1)/2.0 - (fV2 - fV1)/ 4.0;
fV_cal1 = fV;
// Calculate acceleration and torque (with/without correction).
fCarA = (( fV - fVbef) / fTm) * 1000.0/(60.0*60.0);                    // Calculates acceleration.

// Calculate engine speed.
nRet = GetNe( nGear, fV, fNe_cal);                                    // Engine speed
if( nRet == NG ){
   return( false );
}
if( fNe_cal < m_fClutch_MeetNe ){
   fNe_cal = m_fClutch_MeetNe;
}
if( fNe_cal >= m_fOutputRotation ){
   fNe_cal = m_fOutputRotation;
   nRet = GetV( nGear, fNe_cal, fV );
   if( nRet == NG ){
      return( false );
   }
   fV_cal1 = fV;
   fCarA = (( fV - fVbef) / fTm) * 1000.0/(60.0*60.0);
}
fNe = fNe_cal;
nRet = GetTe_NotRevise( nGear, fV, fCarA, fNe_cal, fTe_cal);           // Engine torque (without complement)
if( nRet == NG ){
   return( false );
}
fTe_spl = GetLineReviseMaxTorque(fNe_cal);                            // Linear interpolation

if( fTe_spl != 0 ){
   if(( 0 <= ( fTe_spl - fTe_cal) )&&
      ( ( fTe_spl - fTe_cal  ) < 1.0E-6 )){                           // If nearest value is found
       break;
   }
   fTe_DIF_1 = (fTe_cal / fTe_spl );
}else{
```

```
    fTe_DIF_1 = 0;
}


// Since not found, reduce the speed range
// and restart calculation.

// Check that cross point is generated at more than one location.
if( ( fTe_DIF_1 < 1.0 )&&( fTe_DIF_2 < 1.0 )&&( fTe_DIF_M > 1.0 ) ){
    return(false);
}


// First, determination is made for mid-point.
if( fTe_DIF_M > 1.0 ){                                  // If mid-point is still in Te > TeMax
    if( ( 1.0 < fTe_DIF_1 )&&
        ( fTe_DIF_1 < fTe_DIF_M )&&
        ( fTe_DIF_M < fTe_DIF_2 )){                     // If Te1 is near cross point
        // -------------------------
        // If Te1 is near cross point
        // -------------------------
        fV2 = fV_cal1;                                  // Sets range end to Te1 speed.
    }else if( ( fTe_DIF_1 < 1.0 )&&
              ( 1.0 < fTe_DIF_M )&&
              ( fTe_DIF_M < fTe_DIF_2 ) ){              // If cross point is between Te1 and TeM.
        // -------------------------
        // If cross point is between Te1 and TeM
        // -------------------------
        fV1 = fV_cal1;                                  // Sets range start point to Te1 speed.
        fV2 = fV_calM;                                  // Sets range end to TeM speed.
    }else if( ( fTe_DIF_1 > fTe_DIF_M )&&
              ( fTe_DIF_M > 1.0 )&&
              ( 1.0 > fTe_DIF_2 )){                     // Cross point is between TeM and Te2.
        // -------------------------
        // If cross point is between TeM and Te2
        // -------------------------
        fV1 = fV_calM;                                  // Sets range start point to TeM speed.
        fV2 = fV_cal2;                                  // Sets range end to Te2 speed.
    }else if( ( fTe_DIF_1 > fTe_DIF_M )&&
              ( fTe_DIF_M > fTe_DIF_2 )&&
              ( fTe_DIF_2 > 1.0 )){                     // If cross point is near Te2
        // -------------------------
        // If cross point is near Te2
        // -------------------------
        fV1 = fV_cal2;                                  // Sets range start point to Te2 speed.
    }else{
        if(( fTe_DIF_1 == fTe_DIF_M )&&
           ( fTe_DIF_M == fTe_DIF_2 )){
            return(true);
        }
        if( fTe_DIF_1 == fTe_DIF_M ){
            fV1 = fV_calM;
        }else if( fTe_DIF_M == fTe_DIF_2 ){
            fV2 = fV_calM;
        }else{
            return(true);
        }
    }
}else{                                                  // If TeMax > Te
    // If cross point is near mid-point
    if( ( fTe_DIF_1 < fTe_DIF_M )&&
        ( fTe_DIF_M < fTe_DIF_2 )&&
        ( fTe_DIF_2 < 1.0 )){                           // If cross point is near Te2
        // ---------------------------
        // If cross point is near Te2
        // ---------------------------
        fV1 = fV_cal2;                                  // Sets range start point to Te2 speed.
    }else if( ( fTe_DIF_1 < fTe_DIF_M )&&
              ( fTe_DIF_M < 1.0 )&&
              ( 1.0 < fTe_DIF_2 )){                     // If cross point is between TeM and Te2.
        // ---------------------------
        // If cross point is between TeM and Te2
        // ---------------------------
        fV1 = fV_calM;                                  // Sets range start point to TeM speed.
        fV2 = fV_cal2;                                  // Sets range end to Te2 speed.
    }else if( ( fTe_DIF_1 > 1.0 )&&
              ( 1.0 > fTe_DIF_M )&&
```

```
                ( fTe_DIF_M > fTe_DIF_2 )){                    // If cross point is between Te1 and TeM
            // ----------------------------
            // If cross point is between Te1 and TeM
            // ----------------------------
            fV1 = fV_cal1;                                     // Sets range start point to Te1 speed.
            fV2 = fV_calM;                                     // Sets range end to TeM speed.
          }else if( ( 1.0 > fTe_DIF_1 )&&
                ( fTe_DIF_1 > fTe_DIF_M )&&
                ( fTe_DIF_M > fTe_DIF_2 )){                    // If cross point is near Te1
            // -------------------------
            // If cross point is near Te1
            // -------------------------
            fV2 = fV_cal1;                                     // Sets range end to Te1 speed.
          }else{
            if(( fTe_DIF_1 == fTe_DIF_M )&&
              ( fTe_DIF_M == fTe_DIF_2 )){
              return(true);
            }
            if( fTe_DIF_1 == fTe_DIF_M ){
              fV2 = fV_calM;
            }else if( fTe_DIF_M == fTe_DIF_2 ){
              fV1 = fV_calM;
            }else{
              return(true);
            }
          }
        }
      }
      if( i >= 10000 ){
        return( false );
      }
    }

    return( true );
}
/**/
/*****************************************************************
 * Function name          : BtwnTimeSet
 * Function summary       : Section time setup processing
 * Explanation            : Time is set for specified section.
 *                        :
 * Argument (input)       : p_first : First pointer
 * Argument (input)       : p_end   : Final pointer
 * Argument (output)      : None
 * Argument (I/O)         : None
 * Return value           :
 * Created by             :
 * Updated on (created on) :
 * Remarks                :
 *****************************************************************/
void   TCalculateProc::BtwnTimeSet(map<double,stCalculateData>::iterator p_first,
                    map<double,stCalculateData>::iterator p_end )
{
    map<double,stCalculateData>::iterator p_tmp;
    map<double,stCalculateData>::iterator p_next;                  // Next pointer
    // ---------------------------------------------------------
    // Set section time for analysis data.
    // ---------------------------------------------------------
    for( p_tmp = p_first;p_tmp != p_end;p_tmp++ ){                 // Loops for number of analysis data items.
      p_next = p_tmp; p_next++;                                    // Sets next pointer.
      if( p_next != setCalculateData.end() ){                      // Sets next pointer for other than final data.
        p_tmp->second.nCalcTime = (int)(p_next->second.fTimes - p_tmp->second.fTimes); // Sets section using accumulated time.
      }
    }

    return;
}
/**/
/*****************************************************************
 * Function name          : BtwnCarASet
 * Function summary       : Section acceleration setup processing
 * Explanation            : Speed is set for specified section.
 *                        :
 * Argument (input)       : p_first : First pointer
 * Argument (input)       : p_end   : Final pointer
 * Argument (output)      : None
 * Argument (I/O)         : None
```

```
 * Return value           :
 * Created by             :
 * Updated on (created on) :
 * Remarks                :
 ********************************************************************/
void   TCalculateProc::BtwnCarASet(map<double,stCalculateData>::iterator p_first,
                        map<double,stCalculateData>::iterator p_end )
{
    map<double,stCalculateData>::iterator p_tmp;
    map<double,stCalculateData>::iterator p_next;                              // Next pointer
    double  fVx1,fVx2;                                                  // Temporary reference vehicle speed
    double  fTx1,fTx2;                                                  // Temporary accumulated time
    double  fCarA;                                                      // Temporary acceleration

    // ------------------------------------
    // Set acceleration in previous data.
    // ------------------------------------
    if( p_first != setCalculateData.begin() ){
        p_first--;
        p_end--;
    }
    // ----------------------------------------------------------------
    // Set section time for analysis data.
    // ----------------------------------------------------------------
    for( p_tmp = p_first;p_tmp != p_end;p_tmp++ ){                             // Loops for number of analysis data items.
        p_next = p_tmp; p_next++;                                       // Sets next pointer.
        if( p_next != setCalculateData.end() ){                         // Sets next pointer for other than final data.
            fVx1 = p_tmp->second.fVana_sp;                              // Sets first speed.
            fTx1 = p_tmp->second.fTimes /10.0;                          // Sets first time.
            fVx2 = p_next->second.fVana_sp;                             // Sets next point speed.
            fTx2 = p_next->second.fTimes /10.0;                         // Sets next point time.
            if(( fVx2 - fVx1 == 0 )||( fTx2 - fTx1 == 0 )){             // If time is 0 or speed not changed
                fCarA = 0;                                              // Sets acceleration to 0.
            }else{
                fCarA = ( fVx2 - fVx1) / (fTx2 - fTx1);                 // Calculates acceleration.
            }

            // Acceleration setting
            p_tmp->second.fA = fCarA;                                   // Sets acceleration.
            p_tmp->second.fCarA = fCarA * 1000.0/(60.0*60.0);           // _________ km/sec -> m/msec
        }
    }
    return;
}
/**/
/*********************************************************************
 * Function name          : Calculate_progress1
 * Function summary       : Reference speed/reference acceleration setup processing
 * Explanation            : Reference speed and acceleration are calculated and set.
 *                        :
 * Argument (input)       : None
 * Argument (output)      : None
 * Argument (I/O)         : None
 * Return value           : true : Normal    false : Failure
 * Created by             :
 * Updated on (created on) :
 * Remarks                :
 ********************************************************************/
bool   TCalculateProc::Calculate_progress1()
{
    map<double,stCalculateData>::iterator p_first;                             // First data
    map<double,stCalculateData>::iterator p_second;                            // Next data
    double tmpfTimes;                                                   // Temporary accumulated time
    double fVx1,fVx2;                                                   // Temporary reference vehicle speed
    double fTx1,fTx2;                                                   // Temporary accumulated time
    double fCarA;                                                       // Temporary acceleration
    double tmpfV;                                                       // Temporary reference vehicle speed (first position)
    bool   tmpbIdle;                                                    // Temporary IDLE state

    // ------------------------------------
    // End as is if no analysis data exists.
    // ------------------------------------
    if( setCalculateData.empty() == true ){                             // If no analysis data exists
        return( true );
    }
    // ------------------------------------
    // Obtain data from vehicle speed in analysis data
```

```
    // till next vehicle speed.
    // ------------------------------------
    tmpfTimes = 0.0;                                        // Sets temporary accumulated time.
    p_first = setCalculateData.end();                       // Initializes reference position.
    p_second = setCalculateData.end();                      // Initializes next position.
    for( p_setCalculateData = setCalculateData.begin();
              p_setCalculateData != setCalculateData.end();
              p_setCalculateData++ ){                       // Loops for number of analysis data items.
      if(( p_setCalculateData->second.fV == 0 )&&
        ( p_setCalculateData->second.nPtnReadFlg == 0 )){   // If read vehicle speed does not exist and pattern read data does not exist
        if(( p_first != setCalculateData.end() )&&
          ( p_second == setCalculateData.end() )){          // If first point is recognized but next point is not found
          p_setCalculateData->second.fbtwnTime =
              (double)(p_setCalculateData->second.fTimes / 10.0)  -
                        (double)tmpfTimes;                  // Sets elapsed time from first point.
        }
        continue;
      }

      if( p_first == setCalculateData.end() ){              // If first point is not set
        p_first = p_setCalculateData;                       // Sets pointer as first position.
        tmpfTimes = p_setCalculateData->second.fTimes /10.0;        // Sets elapsed time as temporary accumulated time.
        p_setCalculateData->second.fbtwnTime = 0;           // Sets elapsed time from first point as 0.
        continue;
      }

      if( p_second == setCalculateData.end() ){             // If next point is not set
        p_second = p_setCalculateData;                      // Sets pointer as next position.
      }

      // --------------------------------
      // Calculate acceleration.
      // --------------------------------
      fVx1 = p_first->second.fV;                            // Sets first speed.
      fTx1 = p_first->second.fTimes /10.0;                  // Sets first time.
      fVx2 = p_second->second.fV;                           // Sets next point speed.
      fTx2 = p_second->second.fTimes /10.0;                 // Sets next point time.
      if(( fVx2 - fVx1 == 0 )||( fTx2 - fTx1 == 0 )){       // If time is 0 or speed not changed
        fCarA = 0;                                          // Sets acceleration to 0.
      }else{
        fCarA = ( fVx2 - fVx1) / (fTx2 - fTx1);             // Calculates acceleration.
      }
      tmpfV = p_first->second.fV;                           // Sets first speed as temporary speed.
      tmpbIdle = p_first->second.bIdle;                     // Sets temporary IDLE state.
      // --------------------------------
      // Calculate reference speed and reference acceleration.
      // --------------------------------
      for( p_setCalculateData = p_first;
              p_setCalculateData != p_second;
              p_setCalculateData++ ){                       // Loops from first to next positions.
        // Acceleration setting
        p_setCalculateData->second.fA = fCarA;              // Sets acceleration.
        p_setCalculateData->second.fCarA = fCarA * 1000.0/(60.0*60.0);      // ___________ km/sec -> m/msec
        // Reference vehicle speed setting
        p_setCalculateData->second.fVref_sp =
          tmpfV + fCarA * p_setCalculateData->second.fbtwnTime;     // Calculates reference vehicle speed.
        p_setCalculateData->second.bIdle = tmpbIdle;        // Sets temporary IDLE state.
      }

      p_first = p_second;                                   // Sets next point as first position.
      p_second = setCalculateData.end();                    // Sets next point as end position.
      tmpfTimes = p_first->second.fTimes /10.0;             // Sets elapsed time as temporary accumulated time.
      p_setCalculateData->second.fbtwnTime = 0;             // Sets elapsed time from first point as 0.
    }

    return( true );
}
/**/
/*****************************************************************
 * Function name          : Calculate_progress2
 * Function summary       : Processing flag setup processing
 * Explanation            : Processing methods are classified based on speed and acceleration.
 *                        :
 * Argument (input)       : None
 * Argument (output)      : None
 * Argument (I/O)         : None
 * Return value           : true : Normal    false : Failure
```

```
 * Created by              :
 * Updated on (created on) :
 * Remarks                 :
 ************************************************************************/
bool   TCalculateProc::Calculate_progress2()
{
    map<double,stCalculateData>::iterator p_before;                         // Previous pointer
    int    nBefFlag;                                                        // Previous flag
    int    nGear;                                                           // Previous gear position
    double fBef_V;                                                          // Previous speed

    nBefFlag = 0;                                                           // Initializes preceding flag.
    nGear = 0;
    fBef_V = 0;
    for( p_setCalculateData = setCalculateData.begin();
            p_setCalculateData != setCalculateData.end();
            p_setCalculateData++ ){                                         // Executed for all analysis data items
        if( p_setCalculateData->second.fVref_sp == fBef_V ){                // If previous speed is same
            if( nBefFlag == 0 ){                                            // If previous operation is IDLE state.
                p_setCalculateData->second.nFlag = 0;                       // Operation for this time is also IDLE state.
            }else{                                                          // If other than IDLE state
                p_setCalculateData->second.nFlag = 5;                       // Operation for this time is acceleration processing.
            }
        }else if( p_setCalculateData->second.fVref_sp > fBef_V ){           // If faster than previous speed
            if( p_setCalculateData == setCalculateData.begin() ){           // In case of first data
                p_setCalculateData->second.nFlag = 2;                       // Executes constant speed processing.
            }else if( nBefFlag == 0 ){                                      // If previous operation is IDLE state
                p_before->second.nFlag = 5;                                 // Changes previous flag to Start as well.
                p_setCalculateData->second.nFlag = 5;                       // Operation for this time is acceleration state.
            }else{
                p_setCalculateData->second.nFlag = 5;                       // Operation for this time is acceleration state.
            }
        }else{                                                              // If slower than previous speed
            if( p_setCalculateData->second.fVref_sp == 0 ){                 // If speed is 0
                p_setCalculateData->second.nFlag = 0;                       // Sets to IDLE state for this time.
            }else{
                p_setCalculateData->second.nFlag = 4;                       // Sets to deceleration state for this time.
            }
        }
        nGear = p_setCalculateData->second.nGear;                           // Holds gear position for this time.
        nBefFlag = p_setCalculateData->second.nFlag;                        // Holds flag for this time.
        fBef_V = p_setCalculateData->second.fVref_sp;                       // Holds speed for this time.
        p_before = p_setCalculateData;                                      // Sets previous pointer.
    }
    return( true );

}

/**/
/************************************************************************
 * Function name          : Calculate_progress3
 * Function summary       : Processing data setup processing
 * Explanation            : According to each processing method, applicable module is initiated
 *                        : and data is set.
 * Argument (input)       : None
 * Argument (output)      : None
 * Argument (I/O)         : None
 * Return value           : true : Normal    false : Failure
 * Created by             :
 * Updated on (created on) :
 * Remarks                :
 ************************************************************************/
bool   TCalculateProc::Calculate_progress3()
{
    map<double,stCalculateData>::iterator p_first;                          // First data
    map<double,stCalculateData>::iterator p_second;                         // Next data
    map<double,stCalculateData>::iterator p_betwn;                          // Identifies pointers before and after.
    bool bRet;                                                              // Function return value
    int  tmpSize, tmpNow;                                                   // Percentage
    char buf[256];

    tmpSize = (int)setCalculateData.size();                                 // Sets size.
    tmpNow = 0;

    // ------------------------------------------------
    // Make gear position settings for all analysis data items.
    // ------------------------------------------------
```

```
for( p_setCalculateData = setCalculateData.begin();
      p_setCalculateData != setCalculateData.end();
      p_setCalculateData++ ){                                  // Checks all analysis data items.


   // ------------------------
   // Determine target range.
   // ------------------------
   p_first = p_setCalculateData;                               // Sets first range position.
   for( p_second = p_first; p_second->second.nFlag == p_first->second.nFlag;
                        p_second++ ){                          // Up to same flag
      tmpNow++;                                                // Increments count.
      if( p_second == setCalculateData.end() ){
         break;
      }
   }

   sprintf( buf, "\b\b\b\b\b\b%5.1f%%", (double)tmpNow / (double)tmpSize * 100.0 );
   cout << buf;
   if( tmpSize == tmpNow ){
      cout << "\b\b\b\b\b\b      ";
      cout << endl;
   }

   switch( p_setCalculateData->second.nFlag ){                 // Sets gear according to pattern information flag.
      case 0:                                                  // In case of IDEL state
         bRet = Calculate_SetIDLE( p_first,p_second );         // Sets IDLE state.
         if( bRet == false ){
            return( false );
         }
         p_setCalculateData = p_second;                        // Sets IDLE state end position.
         p_setCalculateData--;                                 // ____________________
         break;
      case 1:                                                  // Starting gear setting
         bRet = Calculate_Set_Start( p_first, p_second );      // Sets starting gear position.
         if( bRet == false ){
            return( false );
         }
         p_setCalculateData = p_second;                        // Increments count until vehicle start
         bRet = Calculate_Start_Following(p_setCalculateData );// Post-processing for vehicle start
         if( bRet == false ){
            return( false );
         }
         break;
      case 2:                                                  // Gear setting for constant speed running
         bRet = Calculate_Set_SteadyState( p_first, p_second );// Sets gear position for constant speed running.
         if( bRet == false ){
            return( false );
         }
         p_setCalculateData = p_second;                        // Increments count until deceleration end.
         p_setCalculateData--;                                 // _______________
         break;
      case 3:                                                  // Gear setting for stop processing (clutch disengaged)
         break;
      case 4:                                                  // Gear setting for deceleration
         bRet = Calculate_T6Set(p_first, p_second );           // Sets free-running time for deceleration.
         if( bRet == false ){
            return( false );
         }
         bRet = Calculate_Set_Deceleration( p_first, p_second );// Sets gear position for deceleration.
         if( bRet == false ){
            return( false );
         }
         p_setCalculateData = p_second;                        // Increments count until deceleration end.
         p_setCalculateData--;                                 // _______________
         break;
      case 5:                                                  // Gear setting for acceleration
         bRet = Calculate_T3Set(p_first, p_second );           // Sets free-running time for acceleration.
         if( bRet == false ){
            return( false );
         }
         bRet = Calculate_Set_Acceleration( p_first, p_second );// Sets gear position for acceleration.
         if( bRet == false ){
            return( false );
         }
         p_setCalculateData = p_second;                        // Increments count until acceleration end.
         p_setCalculateData--;                                 // _______________
```

```
            break;
        }

    }

    cout << "\b\b\b\b\b\b      " << endl;

    return( true );
}
/**/
/*****************************************************************
 * Function name          : Calculate_SetIDLE
 * Function summary       : Idle section setup processing
 * Explanation            : Settings are made for idling section.
 *                        :
 * Argument (input)       : p_first  : First pointer
 * Argument (input)       : p_second : Next setting pointer
 * Argument (output)      : None
 * Argument (I/O)         : None
 * Return value           : true : Normal    false : Failure
 * Created by             :
 * Updated on (created on) :
 * Remarks                :
 *****************************************************************/
bool   TCalculateProc::Calculate_SetIDLE( map<double,stCalculateData>::iterator p_first,
                                          map<double,stCalculateData>::iterator p_second )
{
    map<double,stCalculateData>::iterator p_before;                         // Data before section processing
    double  fNegrevo;                                                       // Engine speed
    double  fTe;                                                            // Engine torque
    double  fRL;                                                            // R/L rolling resistance

    if( p_first != setCalculateData.begin() ){
        p_before = p_first; p_before--;                                     // Sets previous value.
    }
    while( p_first != p_second ){                                           // Loops according to range.
        fNegrevo = m_fldleNe;                                               // Engine speed
        fRL = CalcRL(0);                                                    // Rolling resistance
        fTe = 0;                                                            // Engine torque


        p_first->second.fNegrevo = fNegrevo;                                // Sets engine speed.
        p_first->second.fRL = fRL;                                          // Sets load factor.
        p_first->second.fTe = fTe;                                          // Sets engine torque.
        p_first->second.bldle = true;                                       // Sets IDLE state.
        p_first++;                                                          // Next position processing
    }
    return( true );
}
/**/
/*****************************************************************
 * Function name          : Calculate_Set_Start
 * Function summary       : Starting section setup processing
 * Explanation            : Settings are made for starting section.
 *                        :
 * Argument (input)       : p_first  : First pointer
 * Argument (input)       : p_second : Next setting pointer
 * Argument (output)      : None
 * Argument (I/O)         : None
 * Return value           : true : Normal    false : Failure
 * Created by             :
 * Updated on (created on) :
 * Remarks                :
 *****************************************************************/
bool   TCalculateProc::Calculate_Set_Start( map<double,stCalculateData>::iterator p_first,
                                          map<double,stCalculateData>::iterator p_second )
{
    map<double,stCalculateData>::iterator p_before;                         // Data before section processing
    int befGear;                                                            // Previous gear position
    int befGearTime;                                                        // Previous gear determination time
    int nCount;                                                             // Counter

    double fCarA;                                                           // Acceleration
    double  fVana_sp;                                                       // Analysis vehicle speed
    double      fVref_sp;                                                   // Reference vehicle speed
    double  fNegrevo;                                                       // Engine speed
    double  fTe;                                                            // Engine torque
```

```
double  fRL;                                        // R/L rolling resistance

// --------------------
// Initialization
// --------------------
fTe = 0.0;
fRL = 0.0;

p_before = p_first;
if( p_before != setCalculateData.begin() ){                 // If other than immediate execution of start processing
   p_before--;                                      // Uses preceding data.
   befGearTime = p_before->second.nGearTime;                // Previous gear setup time
}else{
   befGearTime = 0;
}

// Use previous data.
if( p_before->second.nFlag != 0 ){
   fCarA    = p_before->second.fCarA;                  // Previous acceleration
   fVana_sp = p_before->second.fVana_sp;                // Previous analysis speed
   fVref_sp = p_before->second.fVref_sp;                // Previous reference vehicle speed
   befGear = p_before->second.nGear;
}else{                                              // In case of immediate execution of starting
   if( p_before != setCalculateData.begin() ){              // If other than immediate execution of start processing
      fCarA    = p_before->second.fCarA;                 // Previous acceleration
      fVana_sp = p_before->second.fVana_sp;              // Previous analysis speed
      fVref_sp = p_before->second.fVref_sp;              // Previous reference vehicle speed
      befGear = p_before->second.nGear;                 // Previous gear position
   }else{
      fCarA = 0;                                    // Acceleration = 0
      fVana_sp = 0;                                 // Analysis speed = 0
      fVref_sp = 0;                                 // Reference vehicle speed = 0
      befGear = 0;                                  // Previous gear position = 0
   }
}

// ------------------------------
// Set IDLE engine speed as initial value.
// ------------------------------
fNegrevo = m_fIdleNe;
nCount = 0;
if( p_first != p_second ){                          // In case of other than first section
   while( p_first != p_second ){                    // Loops according to range.
      if( p_first->second.nTimeFlg == 2 ){              // t2 section (time to reach starting engine speed)
         fNegrevo = m_fIdleNe;                      // Engine speed
         fRL = CalcRL(0);                           // Rolling resistance
         fTe = 0;                                   // Engine torque
      }

      // --------------------------------
      // Set calculated analysis data.
      // --------------------------------
      p_first->second.fVref_sp = fVref_sp;              // Sets reference speed.
      p_before->second.fCarA = fCarA;                   // Sets acceleration.
      p_first->second.fNegrevo = fNegrevo;              // Sets engine speed.
      p_first->second.fRL = fRL;                        // Sets load factor.
      p_first->second.fTe = fTe;                        // Sets engine torque.
      p_first->second.nGear = 0;
      befGearTime = 0;


      nCount++;                                     // Increments counter.
      p_first++;                                    // Next position processing
      p_before = p_first;
      p_before--;                                   // Processing of previous position
      if( p_first != setCalculateData.end() ){              // In case of other than final data
         p_first->second.nGearTime = befGearTime;           // Sets gear setup time.
      }
   }
}else{                                              // In case of first section
   p_first->second.fCarA = fCarA;                       // Sets acceleration.
   p_first->second.fNegrevo = fNegrevo;                 // Sets engine speed.
   p_first->second.fRL = 0.0;                           // Sets load factor.
   p_first->second.fTe = 0.0;                           // Sets engine torque.

   p_first->second.nGear = 0;
   befGearTime = 0;
```

```
        p_first++;                                                  // Next position processing
        if( p_first != setCalculateData.end() ){                    // In case of other than final data
            p_first->second.nGearTime = befGearTime;                // Sets gear setup time.
        }
    }

    return( true );
}
/**/
/*****************************************************************
 * Function name          : Calculate_Set_SteadyState
 * Function summary       : Constant speed section setup processing
 * Explanation            : Settings are made for constant speed section.
 *                        :
 * Argument (input)       : p_first  : First pointer
 * Argument (input)       : p_second : Next setting pointer
 * Argument (output)      : None
 * Argument (I/O)         : None
 * Return value           : true : Normal    false : Failure
 * Created by             :
 * Updated on (created on) :
 * Remarks                :
 *****************************************************************/
bool   TCalculateProc::Calculate_Set_SteadyState( map<double,stCalculateData>::iterator p_first,
                                    map<double,stCalculateData>::iterator p_second )
{
    map<double,stCalculateData>::iterator p_before;                 // Data before section processing
    int        nRet;                                                // Function return value
    int        nGear;                                               // Gear position for section setting
    int        befGear;                                             // Previous gear position
    double     befGearTime;                                         // Previous gear hold time
    double     befCalcTime;                                         // Previous required time
    double     fCarA;                                               // Acceleration
    double     fVana_sp;                                            // Analysis vehicle speed
    double     fVref_sp;                                            // Reference vehicle speed
    double     fVana_sp0;                                           // Analysis vehicle speed
    double     fVref_sp0;                                           // Reference vehicle speed
    double     fNegrevo;                                            // Engine speed
    double     fTe;                                                 // Engine torque
    double     fRL;                                                 // R/L rolling resistance

    // --------------------
    // Initialization
    // --------------------
    p_before = p_first;
    fCarA    = p_first->second.fCarA;                               // Acceleration for this time
    if( p_before != setCalculateData.begin() ){                     // In case of other than first time
        p_before--;                                                 // Uses preceding data.
        befGear = p_before->second.nGear;                           // Previous gear position
        befGearTime = p_before->second.nGearTime;                   // Previous gear hold time
        befCalcTime = p_before->second.nCalcTime;                   // Previous required time
        fVana_sp = p_before->second.fVana_sp;                       // Analysis speed for this time (previous value as default)
        fVref_sp = p_before->second.fVref_sp;                       // Reference vehicle speed for this time (previous value as default)
        fVana_sp0 = p_before->second.fVana_sp;                      // Previous analysis speed
        fVref_sp0 = p_before->second.fVref_sp;                      // Previous reference vehicle speed
        nGear     = p_before->second.nGear;                         // Previous gear position
    }else{
        befGear = p_before->second.nGear;                           // Previous gear position
        befGearTime = p_before->second.nGearTime;                   // Previous gear hold time
        befCalcTime = p_before->second.nCalcTime;                   // Previous required time
        fVana_sp = p_before->second.fVana_sp;                       // Analysis speed for this time (previous value as default)
        fVref_sp = p_before->second.fVref_sp;                       // Reference vehicle speed for this time (previous value as default)
        fVana_sp0 = p_before->second.fVana_sp;                      // Previous analysis speed
        fVref_sp0 = p_before->second.fVref_sp;                      // Previous reference vehicle speed
        nGear     = p_before->second.nGear;                         // Previous gear position
    }

    if( p_first != p_second ){                                      // In case of other than first section
        while( p_first != p_second ){                               // Loops according to range.
            p_before = p_first;p_before--;
            fCarA = p_before->second.fCarA;

            // ------------------------------------------------
            // Calculate speed for this time based on previous analysis vehicle speed and acceleration.
            // ------------------------------------------------
            fVref_sp = p_first->second.fVref_sp;                    // Sets reference vehicle speed.
```

```
        fVana_sp = fVana_sp + fCarA * befCalcTime /10.0 * 3.6;                    // Analysis vehicle speed

        fRL = CalcRL(fVana_sp);                                                   // R/L rolling resistance

        nRet = GetNe( nGear, fVana_sp, fNegrevo);                                 // Engine speed
        if (nRet == NG) return( false );

        if( fNegrevo < m_fClutch_ReleaseNe ){                                     // In case of smaller than clutch release engine speed
           p_first->second.nFlag = 2;
           p_first->second.bIdle = true;
           p_first->second.nGear = 0;
           fNegrevo = m_fClutch_ReleaseNe;
           fTe = 0;                                                               // Engine torque
           p_first->second.nGearTime = 0;                                         // Clears gear setting time.
        }else{
           nRet = GetTe( nGear,fVana_sp, fCarA,fNegrevo, fTe);                    // Engine torque
           if (nRet == NG) return( false );
           p_first->second.nGearTime = (int)befGearTime +
                       p_first->second.nCalcTime;                                // Gear setup time addition setting
        }


        p_first->second.nGearTime = (int)befGearTime +
                    p_first->second.nCalcTime;                                   // Gear setup time addition setting

        // --------------------------------
        // Set calculated analysis data.
        // --------------------------------
        p_first->second.fVana_sp = fVana_sp;                                     // Sets analysis vehicle speed.
        p_first->second.fNegrevo = fNegrevo;                                     // Sets engine speed.
        p_first->second.fRL = fRL;                                               // Sets load factor.
        p_first->second.fTe = fTe;                                               // Sets engine torque.
        p_first->second.nGear   = nGear;                                         // Sets gear position.
        // ------------------------------------
        // Set data for calculation below.
        // ------------------------------------
        befGear     = p_first->second.nGear;                                     // Previous gear position
        befGearTime = p_first->second.nGearTime;                                 // Previous gear hold time
        befCalcTime = p_first->second.nCalcTime;                                 // Previous required time
        fVana_sp = p_first->second.fVana_sp;                                     // Previous analysis speed
        fVref_sp = p_first->second.fVref_sp;                                     // Previous reference vehicle speed
        fVana_sp0 = p_first->second.fVana_sp;                                    // Previous analysis speed
        fVref_sp0 = p_first->second.fVref_sp;                                    // Previous reference vehicle speed


        if(  p_first == p_second  ){
           break;
        }
        p_before = p_first;
        p_first++;                                                               // Next position processing
     }
   }else{                                                                        // In case of first section
        p_first->second.fCarA = fCarA;                                           // Sets acceleration.
        p_first->second.fNegrevo = 0.0; // fNegrevo;                             // Sets engine speed.
        p_first->second.fRL = 0.0; // fRL;                                       // Sets load factor.
        p_first->second.fTe = 0.0; // fTe;                                       // Sets engine torque.
        if( p_first->second.nGear == 0 ){                                        // If no gear is set
           p_first->second.nGear   = nGear;                                      // Sets gear position.
        }
        befGearTime = befGearTime +
                    p_first->second.nCalcTime;                                   // Sets gear setup time.
        p_first++;                                                               // Next position processing
        if( p_first != setCalculateData.end() ){                                 // In case of other than final data
           p_first->second.nGearTime = (int)befGearTime;                         // Sets gear setup time.
        }
   }

   return( true );
}
/**/
/*****************************************************************
 * Function name          : Calculate_Set_Deceleration
 * Function summary       : Deceleration section setup processing
 * Explanation            : Settings are made for deceleration section.
 *                        :
 * Argument (input)       : p_first  : First pointer
 * Argument (input)       : p_second : Next setting pointer
```

```
 * Argument (output)       : None
 * Argument (I/O)          : None
 * Return value            : true : Normal    false : Failure
 * Created by              :
 * Updated on (created on) :
 * Remarks                 :
 ************************************************************************/
bool   TCalculateProc::Calculate_Set_Deceleration( map<double,stCalculateData>::iterator p_first,
                                    map<double,stCalculateData>::iterator p_second )
{
   map<double,stCalculateData>::iterator p_before;                          // Data before section processing
   map<double,stCalculateData>::iterator p_Next;                            // Next data
   int        nRet;                                                      // Function return value
   int        nGear;                                                     // Gear position for section setting
   double     befGearTime;                                                  // Previous gear hold time
   double     befCalcTime;                                                 // Previous required time
   int        nCount;                                                    // Counter
   double     fCarA;                                                       // Acceleration
   double     fVana_sp,calcVana;                                             // Analysis vehicle speed
   double     fVref_sp,calcVref;                                           // Reference vehicle speed
   double     fVana_sp0;                                                   // Analysis vehicle speed
   double     fVref_sp0;                                                   // Reference vehicle speed
   double     fNegrevo;                                                    // Engine speed
   double     fTe;                                                       // Engine torque
   double     fRL;                                                       // R/L rolling resistance

   // --------------------
   // Initialization
   // --------------------
   // Idling engine speed
   p_before = p_first;
   fCarA    = p_first->second.fCarA;                                     // Acceleration for this time
   nGear    = p_before->second.nGear;                                      // Gear position for this time
   if( p_before != setCalculateData.begin() ){                             // In case of other than first time
      p_before--;                                                       // Uses preceding data.
      befGearTime = p_before->second.nGearTime;                              // Previous gear hold time
      befCalcTime = p_before->second.nCalcTime;                              // Previous required time
      fVana_sp = p_before->second.fVana_sp;                                // Analysis speed for this time (previous value as default)
      fVref_sp = p_before->second.fVref_sp;                                // Reference vehicle speed for this time (previous value as default)
      fVana_sp0 = p_before->second.fVana_sp;                                // Previous analysis speed
      fVref_sp0 = p_before->second.fVref_sp;                               // Previous reference vehicle speed
   }else{
      befGearTime = 0;                                                    // Previous gear hold time
      befCalcTime = 0;                                                    // Previous required time
      fVana_sp = 0;                                                      // Previous analysis vehicle speed
      fVref_sp = 0;                                                      // Previous reference vehicle speed
      fVref_sp0 = 0;                                                     // _______
      fVana_sp0 = 0;                                                     // _______
   }

   // ------------------------------
   // Set IDLE engine speed as initial value.
   // ------------------------------
   fNegrevo = m_fldleNe;
   nCount = 0;
   if( p_first != p_second ){                                             // In case of other than first section
      while( p_first != p_second ){                                       // Loops according to range.
         nGear = p_first->second.nGear;                                      // Sets gear position for this time.

         if( p_first != setCalculateData.begin() ){                          // In case of other than first time
            p_before = p_first;
            p_before--;                                                 // Uses preceding data.
            befGearTime = p_before->second.nGearTime;                          // Previous gear hold time
            befCalcTime = p_before->second.nCalcTime;                          // Previous required time
            fVana_sp = p_before->second.fVana_sp;                            // Analysis speed for this time (previous value as default)
            fVref_sp = p_before->second.fVref_sp;                            // Reference vehicle speed for this time (previous value as default)
            fVana_sp0 = p_before->second.fVana_sp;                            // Previous analysis speed
            fVref_sp0 = p_before->second.fVref_sp;                           // Previous reference vehicle speed
         }
         fCarA = p_before->second.fCarA;

         calcVref = p_first->second.fVref_sp;                               // Sets reference vehicle speed.
         calcVana = fVana_sp + fCarA * befCalcTime /10.0 * 3.6;                  // Analysis vehicle speed
         fCarA = ( calcVref - fVana_sp) /
                              (befCalcTime / 10.0);                     // Calculates acceleration.
         fCarA = fCarA * 1000.0/(60.0*60.0);                                 // Calculates acceleration.
         p_before->second.fCarA = fCarA;                                     // Modifies previous acceleration.
```

```
        // ------------------------------------------------
        // Calculate speed for this time based on previous analysis vehicle speed and acceleration.
        // ------------------------------------------------
        fVref_sp = p_first->second.fVref_sp;                    // Sets reference vehicle speed.
        fVana_sp = fVana_sp + fCarA * befCalcTime /10.0 * 3.6;          // Analysis vehicle speed

        fRL = CalcRL(fVana_sp);                                 // R/L rolling resistance

        nRet = GetNe( nGear, fVana_sp, fNegrevo);                   // Engine speed
        if (nRet == NG) return( false );

        if( fNegrevo < m_fClutch_ReleaseNe ){                   // If smaller than clutch release engine speed
          p_first->second.nFlag = 0;
          p_first->second.bIdle = true;
          p_first->second.nGear = 0;
          fNegrevo = m_fIdleNe;
          fTe = 0;                                              // Engine torque
          p_first->second.nGearTime = 0;                        // Clears gear time.
        }else{
          nRet = GetTe( nGear,fVana_sp, fCarA,fNegrevo, fTe);           // Engine torque
          if (nRet == NG) return( false );

          p_first->second.nGearTime = (int)befGearTime +
                  p_first->second.nCalcTime;                    // Gear setup time addition setting
        }


        // --------------------------------
        // Set calculated analysis data.
        // --------------------------------
        p_first->second.fVana_sp = fVana_sp;                    // Sets analysis vehicle speed.
        p_first->second.fNegrevo = fNegrevo;                    // Sets engine speed.
        p_first->second.fRL = fRL;                              // Sets load factor.
        p_first->second.fTe = fTe;                              // Sets engine torque.
        // ------------------------------------
        // Set data for calculation below.
        // ------------------------------------
        befGearTime = p_first->second.nGearTime;                // Previous gear hold time
        befCalcTime = p_first->second.nCalcTime;                // Previous required time
        fVana_sp = p_first->second.fVana_sp;                    // Previous analysis speed
        fVref_sp = p_first->second.fVref_sp;                    // Previous reference vehicle speed
        fVana_sp0 = p_first->second.fVana_sp;                   // Previous analysis speed
        fVref_sp0 = p_first->second.fVref_sp;                   // Previous reference vehicle speed

        nCount++;                                               // Increments counter.
        p_before = p_first;
        p_first++;                                              // Next position processing
      }
    }else{                                                      // In case of first section
        p_first->second.fCarA = fCarA;                          // Sets acceleration.
        p_first->second.fNegrevo = fNegrevo;                    // Sets engine speed.
        p_first->second.fRL = 0.0; // fRL;                      // Sets load factor.
        p_first->second.fTe = 0.0; // fTe;                      // Sets engine torque.
        if( p_first->second.nGear == 0 ){                       // If no gear is set
          p_first->second.nGear = nGear;                        // Sets gear position.
        }
        befGearTime = befGearTime +
                  p_first->second.nCalcTime;                    // Sets gear setup time.
        p_first++;                                              // Next position processing
        if( p_first != setCalculateData.end() ){                // In case of other than final data
          p_first->second.nGearTime = (int)befGearTime;         // Sets gear setup time.
        }
    }

    return( true );
}
/**/
/*****************************************************************
 * Function name          : Calculate_Set_Acceleration
 * Function summary       : Acceleration section setup processing
 * Explanation            : Settings are made for acceleration section.
 *                        :
 * Argument (input)       : p_first  : First pointer
 * Argument (input)       : p_second : Next setting pointer
 * Argument (output)      : None
 * Argument (I/O)         : None
```

```
 * Return value            : true : Normal    false : Failure
 * Created by              :
 * Updated on (created on) :
 * Remarks                 :
 ********************************************************************/
bool   TCalculateProc::Calculate_Set_Acceleration( map<double,stCalculateData>::iterator p_first,
                                    map<double,stCalculateData>::iterator p_second )
{
    map<double,stCalculateData>::iterator p_before;                         // Data before section processing
    int        nRet;                                                    // Function return value
    bool       bRet;                                                    // Function return value
    int        nGear;                                                   // Gear position for section setting
    int        befGear;                                                 // Previous gear position
    double     befGearTime;                                                // Previous gear hold time
    double     befCalcTime;                                                // Previous required time
    int        nCount;                                                  // Counter
    double     fCarA;                                                    // Acceleration
    double     fVana_sp,calcVana;                                           // Analysis vehicle speed
    double     fVref_sp,calcVref;                                           // Reference vehicle speed
    double     fVana_sp0;                                                 // Analysis vehicle speed
    double     fVref_sp0;                                                 // Reference vehicle speed
    double     fNegrevo;                                                  // Engine speed
    double     fTe;                                                     // Engine torque
    double     fRL;                                                     // R/L rolling resistance

    // --------------------
    // Initialization
    // --------------------
    p_before = p_first;
    fCarA    = p_first->second.fCarA;                                       // Acceleration for this time
    nGear    = p_before->second.nGear;                                        // Gear position for this time
    if( p_before != setCalculateData.begin() ){                               // In case of other than first time
        p_before--;                                                     // Uses preceding data.
        befGear = p_before->second.nGear;                                    // Previous gear position
        befGearTime = p_before->second.nGearTime;                               // Previous gear hold time
        befCalcTime = p_before->second.nCalcTime;                               // Previous required time
        fVana_sp = p_before->second.fVana_sp;                                 // Analysis speed for this time (previous value as default)
        fVref_sp = p_before->second.fVref_sp;                                // Reference vehicle speed for this time (previous value as default)
        fVana_sp0 = p_before->second.fVana_sp;                                 // Previous analysis speed
        fVref_sp0 = p_before->second.fVref_sp;                                // Previous reference vehicle speed
    }else{
        befGear     = 0;                                                  // Previous gear position
        befGearTime = 0;                                                    // Previous gear hold time
        befCalcTime = 0;                                                    // Previous required time
        fVana_sp = 0;                                                    // Previous analysis vehicle speed
        fVref_sp = 0;                                                    // Previous reference vehicle speed
        fVref_sp0 = 0;                                                   // _______
        fVana_sp0 = 0;                                                   // _______
    }


    // --------------------------------
    // Set IDLE engine speed as initial value.
    // --------------------------------
    fNegrevo = m_fIdleNe;
    nCount = 0;
    if( p_first != p_second ){                                              // In case of other than first section
        while( p_first != p_second ){                                         // Loops according to range.
            nGear = p_first->second.nGear;                                       // Sets gear position for this time.

            if( p_first != setCalculateData.begin() ){                           // In case of other than first time
                p_before = p_first;
                p_before--;                                                 // Uses preceding data.
                befGear = p_before->second.nGear;                                // Previous gear position
                befGearTime = p_before->second.nGearTime;                           // Previous gear hold time
                befCalcTime = p_before->second.nCalcTime;                           // Previous required time
                fVana_sp = p_before->second.fVana_sp;                             // Analysis speed for this time (previous value as default)
                fVref_sp = p_before->second.fVref_sp;                            // Reference vehicle speed for this time (previous value as default)
                fVana_sp0 = p_before->second.fVana_sp;                             // Previous analysis speed
                fVref_sp0 = p_before->second.fVref_sp;                            // Previous reference vehicle speed
            }
            fCarA = p_before->second.fCarA;

            if( befGear == 0 ){                                              // Due to no acceleration available with gear position 0
                befGear = nGear;                                               // sets current gear position.
            }
```

```
        // ------------------------------------------------
        // Check max. acceleration based on previous analysis speed, gear, and engine speed.
        // ------------------------------------------------
        calcVref = p_first->second.fVref_sp;                    // Sets reference vehicle speed to prepare for calculation.
        calcVana = p_first->second.fVref_sp;
        bRet = CalcTeMaxSp(nGear, (befCalcTime / 10.0), fVana_sp, calcVana, fNegrevo );
        if( bRet == false ){
          return( false );
        }
        fCarA = (( calcVana - fVana_sp) / (befCalcTime / 10.0)) * 1000.0/(60.0*60.0);  // Calculates acceleration.

        p_before->second.fCarA = fCarA;                         // Modifies previous acceleration.
        // ------------------------------------------------
        // Calculate speed for this time based on previous analysis vehicle speed and acceleration.
        // ------------------------------------------------
        fVref_sp = p_first->second.fVref_sp;                    // Sets reference vehicle speed.
        fVana_sp = calcVana;                                    // Analysis vehicle speed
        fRL = CalcRL(fVana_sp);                                 // R/L rolling resistance

        // ------------------------------------------------
        // Separate cases between free-running time and other.
        // ------------------------------------------------
        if( p_first->second.nTimeFlg == 3 ){                    // In case of t3 section (free-running time during shift-up)
          if(fNegrevo < m_fClutch_ReleaseNe ){                  // If smaller than clutch release engine speed
            fNegrevo = m_fClutch_ReleaseNe;
          }
          p_first->second.nGearTime = 0;                        // Clears gear setup time.
        }else{
          if( fNegrevo < m_fClutch_MeetNe ){
            fNegrevo = m_fClutch_MeetNe;
          }
          p_first->second.nGearTime = (int)befGearTime +
                    p_first->second.nCalcTime;                  // Gear setup time addition setting
        }
        nRet = GetTe( nGear,fVana_sp, fCarA,fNegrevo, fTe);             // Engine torque
        if (nRet == NG){
          return( false );
        }

        // --------------------------------
        // Set calculated analysis data.
        // --------------------------------
        p_first->second.fVana_sp = fVana_sp;                    // Sets analysis vehicle speed.
        p_first->second.fNegrevo = fNegrevo;                    // Sets engine speed.
        p_first->second.fRL = fRL;                              // Sets load factor.
        p_first->second.fTe = fTe;                              // Sets engine torque.
        // ------------------------------------
        // Set data for calculation below.
        // ------------------------------------
        befGear     = p_first->second.nGear;                    // Previous gear position
        befGearTime = p_first->second.nGearTime;                    // Previous gear hold time
        befCalcTime = p_first->second.nCalcTime;                    // Previous required time
        fVana_sp = p_first->second.fVana_sp;                    // Previous analysis speed
        fVref_sp = p_first->second.fVref_sp;                    // Previous reference vehicle speed
        fVana_sp0 = p_first->second.fVana_sp;                   // Previous analysis speed
        fVref_sp0 = p_first->second.fVref_sp;                   // Previous reference vehicle speed

        nCount++;                                               // Increments counter.
        p_before = p_first;
        p_first++;                                              // Next position processing

        if(( p_first == p_second )&&
          ( fVana_sp < fVref_sp )){
          p_second++;
          p_first->second.nFlag = 5;
        }
      }
    }else{                                                      // In case of first section
      p_first->second.fCarA = fCarA;                            // Sets acceleration.
      p_first->second.fNegrevo = fNegrevo;                      // Sets engine speed.
      p_first->second.fRL = 0.0; // fRL;                        // Sets load factor.
      p_first->second.fTe = 0.0; // fTe;                        // Sets engine torque.
      if( p_first->second.nGear == 0 ){                         // If no gear position is set
        p_first->second.nGear   = nGear;                        // Sets gear position.
      }
      befGearTime = befGearTime +
                    p_first->second.nCalcTime;                  // Sets gear setup time.
```

```
        p_first++;                                                  // Next position processing
        if( p_first != setCalculateData.end() ){                    // In case of other than final data
            p_first->second.nGearTime = (int)befGearTime;           // Sets gear setup time.
        }
    }

    return( true );
}
/**/
// ###########################################################################
// ---------------------------------------------------------------------------
// Section setup processing starts here.
// Vehicle starting processing      T1T2Set
// Shift-up  T3Set
// ---------------------------------------------------------------------------
// ###########################################################################
/*****************************************************************
 * Function name          : Calculate_T1T2Set
 * Function summary       : Starting T1-T2 section setup processing
 * Explanation            : Detailed setup processing is executed for starting section.
 *                        :
 * Argument (input)       : None
 * Argument (output)      : None
 * Argument (I/O)         : None
 * Return value           : true : Normal    false : Failure
 * Created by             :
 * Updated on (created on) :
 * Remarks                :
 *****************************************************************/
bool   TCalculateProc::Calculate_T1T2Set()
{
    map<double,stCalculateData>::iterator p_first;                  // First data
    map<double,stCalculateData>::iterator p_second;                 // Next data
    map<double,stCalculateData>::iterator p_betwn;                  // Identifies pointers before and after.
    stCalculateData tmpCalculateData;                               // Temporary analysis data
    int tmpnTimeFlg;                                                // Temporary t1-t6 flag
    int tmpBeforeFlag;                                              // Preceding flag

    p_first = setCalculateData.begin();                             // Sets first data.
    p_second = setCalculateData.end();                              // Sets end position.
    tmpBeforeFlag = 0;                                              // Sets preceding flag.

    // ----------------------------------
    // t2 section (time to reach starting engine speed),
    // t1-t2 section (time from starting engine speed to acceleration speed (t1-t2))
    // If not in analysis data, make additional settings.
    // ----------------------------------
    for( p_setCalculateData = setCalculateData.begin();
         p_setCalculateData != setCalculateData.end();
         p_setCalculateData++ ){                                    // Checks all analysis data items.

        // ------------------------------------
        // Starting position search
        // ------------------------------------
        if(( tmpBeforeFlag == 0 )&&
           ( p_setCalculateData->second.nFlag == 5 )&&
           ( p_setCalculateData->second.nTimeFlg == 0 )){           // If starting position is found
            memset( &tmpCalculateData, 0x00, sizeof( tmpCalculateData ));   // Initializes temporary data.
            // -------------------
            // Set t2 section.
            // -------------------
            tmpCalculateData.fTimes = p_setCalculateData->second.fTimes;    // Obtains position by subtracting T1 (time required for starting)
                                                                    // from time vehicle speed reaches acceleration.
            tmpCalculateData.nCalcTime  = (int)(0.0);               // Section is t2 sec. only.
            tmpCalculateData.bIdle   = false;                       // IDLE state cannot be entered.
            tmpCalculateData.bClutch = true;                        // Clutch engaged state
            tmpCalculateData.nTimeFlg= 2;                           // Sets t2 and time flag.
            tmpCalculateData.nFlag   = 1;                           // Sets flag in starting data.

            p_betwn = setCalculateData.lower_bound( tmpCalculateData.fTimes );  // Checks whether data is within range.
            if( p_betwn != setCalculateData.end() ){                // If data is within range
                if( p_betwn->second.fTimes != tmpCalculateData.fTimes ){    // If data is not added yet
                    setCalculateData.insert(pair<double,stCalculateData>
                                (tmpCalculateData.fTimes, tmpCalculateData) );  // Adds data.
                    p_setCalculateData = setCalculateData.begin();  // Restarts from beginning.
                }else{
                    p_betwn->second.bIdle = false;                  // Idle state OFF
```

```
                p_betwn->second.bClutch = true;                         // Clutch ON
                p_betwn->second.nTimeFlg = 2;                           // Sets T2 flag.
                p_betwn->second.nCalcTime =(int)(0.0);                      // Section is t2 sec. only.
                p_betwn->second.nFlag = 1;                              // Sets flag in starting data.
            }
        }
        memset( &tmpCalculateData, 0x00, sizeof( tmpCalculateData ));           // Initializes temporary data.

    }
    tmpBeforeFlag = p_setCalculateData->second.nFlag;                       // Sets previous flag.
}

// ---------------------------------
// t1-t6 flag hold processing
// ---------------------------------
for( p_setCalculateData = setCalculateData.begin();
     p_setCalculateData != setCalculateData.end();
     p_setCalculateData++ ){                                    // Checks all analysis data items.
    if( p_setCalculateData->second.nTimeFlg == 2 ){                 // If starting position is found
        p_setCalculateData->second.nGearTime = (int)(m_fTg *10);            // Unaware of gear hold for starting
        memset( &tmpCalculateData, 0x00, sizeof( tmpCalculateData ));           // Initializes temporary data.
        // -------------------
        // From start position of t2 section to position with vehicle speed provided
        // -------------------
        tmpCalculateData.fTimes = p_setCalculateData->second.fTimes;            // Obtains position by subtracting T1 (time required for starting)
        p_betwn = setCalculateData.lower_bound( tmpCalculateData.fTimes );      // Checks whether data is within range.
        tmpnTimeFlg = p_setCalculateData->second.nTimeFlg;                  // Holds flag.
        tmpBeforeFlag = p_setCalculateData->second.nFlag;                   // Sets preceding flag.
        for( ; p_betwn != p_setCalculateData; p_setCalculateData++ ){           // Keeps same flag up to section with vehicle speed provided.
            if( p_setCalculateData->second.nTimeFlg == 0 ){                 // If no flag is defined
                p_setCalculateData->second.nTimeFlg = tmpnTimeFlg;              // Holds previous data.
            }
            p_setCalculateData->second.nFlag = tmpBeforeFlag;               // Uses starting data as flag.
            tmpnTimeFlg = p_setCalculateData->second.nTimeFlg;              // Holds flag for this time.

            // ------------------------
            // Set required time as well.
            // ------------------------
            p_second = p_setCalculateData; p_second++;                      // Sets next position.
            if( p_second != setCalculateData.end() ){
                p_setCalculateData->second.nCalcTime = (int)(p_second->second.fTimes -
                            p_setCalculateData->second.fTimes);         // Sets required time from current time.
            }else{                                                      // In case of end of data
                p_setCalculateData->second.nCalcTime = 0;                   // Sets required time to 0.
            }

        }

    }
    // ------------------------
    // Set required time as well.
    // ------------------------
    p_second = p_setCalculateData; p_second++;                          // Sets next position.
    if( p_second != setCalculateData.end() ){
        p_setCalculateData->second.nCalcTime = (int)(p_second->second.fTimes -
                            p_setCalculateData->second.fTimes);         // Sets required time from current time.
    }else{                                                      // In case of end of data
        p_setCalculateData->second.nCalcTime = 0;                       // Sets required time to 0.
    }
}

return( true );
}
/**/
/*****************************************************************
* Function name          : Calculate_Start_Following
* Function summary       : Starting target-speed follow processing
* Explanation            : Settings are made for the case in which target-speed follow is impossible
*                        : during vehicle start.
* Argument (input)       : None
* Argument (output)      : None
* Argument (I/O)         : p_first : First pointer
* Return value           : true : Normal    false : Failure
* Created by             :
* Updated on (created on) :
* Remarks                :
*****************************************************************/
```

```
bool   TCalculateProc::Calculate_Start_Following(map<double,stCalculateData>::iterator &p_first )
{
    map<double,stCalculateData>::iterator p_tmpCalculate;                    // Temporary pointer
    map<double,stCalculateData>::iterator p_second;                          // Next data
    map<double,stCalculateData>::iterator p_betwn;                           // Identifies pointers before and after.
    map<double,stCalculateData>::iterator p_start;                           // Temporary pointer
    int     nRet;                                                  // Return value of function to be recalled
    int     tmpNowGear;                                            // Temporary gear position
    double  tmpVana_sp;                                            // Previous analysis speed
    double  tmpNegrevo;                                            // Previous engine speed
    double  tmpGearTime;                                           // Gear required time
    double  tmpCalcTime;                                           // Required time
    double  fBeforeA;                                              // Previous acceleration
    double  fCarA;                                                 // Previous acceleration
    double  fNegrevo;                                              // Engine speed
    double  fTe;                                                   // Engine torque
    double  fTe_def;                                               // Engine torque
    double  fTeMax;                                                // Engine torque
    double  fRL;                                                   // R/L rolling resistance
    double  fCarV;                                                 // Max. vehicle speed value with clutch in
    double  fGearti;                                               // Gear ratio
    double  calcVana;                                              // Speed data for calculation
    bool    bRet;
    int     nNowGear;                                              // Applicable gear

    p_start = p_first;
    p_tmpCalculate = p_first;
    if( p_first != setCalculateData.begin() ){
        p_tmpCalculate--;                                          // Sets preceding data point.
    }

    tmpNowGear = p_tmpCalculate->second.nGear;                     // Sets preceding gear position.
    if( tmpNowGear == 0 ){
        tmpNowGear = m_nInitGear;
    }
    nNowGear = tmpNowGear;                                         // Applicable gear
    tmpVana_sp = p_tmpCalculate->second.fVana_sp;                  // Previous analysis speed
    tmpNegrevo = p_tmpCalculate->second.fNegrevo;                  // Previous engine speed
    tmpGearTime = p_tmpCalculate->second.nGearTime;                // Previous gear time
    tmpCalcTime = p_tmpCalculate->second.nCalcTime;                // Previous required time
    fBeforeA   = p_tmpCalculate->second.fCarA;                     // Previous acceleration
    // ---------------------------------
    // Determine engine speed availability with current gear position.
    // ---------------------------------
    // Calculate engine speed.
    fNegrevo = m_fIdleNe ;
    nRet = GetGeariN(tmpNowGear,fGearti);
    if (nRet == NG) return( false );
    // Calculate vehicle speed.
    fCarV = (2.0 * CalculateProc->m_fPAI * m_fTarR)/ 1000.0 * 60.0 * fNegrevo
                        / fGearti;                                 // Time for clutch to engage
CHECK_AGAIN:
    p_tmpCalculate = p_first;
    if( p_first != setCalculateData.begin() ){
        p_tmpCalculate--;                                          // Sets preceding data point.
    }

    tmpVana_sp = p_tmpCalculate->second.fVana_sp;                  // Previous analysis speed
    tmpNegrevo = p_tmpCalculate->second.fNegrevo;                  // Previous engine speed
    tmpGearTime = p_tmpCalculate->second.nGearTime;                // Previous gear time
    tmpCalcTime = p_tmpCalculate->second.nCalcTime;                // Previous required time
    fBeforeA   = p_tmpCalculate->second.fCarA;                     // Previous acceleration

    for( ; ; p_first++){                                           // All sections subject to determination
        if(( p_first->second.nFlag != 5 )&&
           ( p_first->second.nFlag != 1 )){
            p_first--;
            break;
        }
        calcVana = p_first->second.fVref_sp;
        fCarA = (( calcVana - tmpVana_sp) / (tmpCalcTime / 10.0)) * 1000.0/(60.0*60.0);
        nRet = GetNe( nNowGear, calcVana, fNegrevo);               // Engine speed
        if( nRet == NG ){
            return( false );
        }
        nRet = GetTe_NotRevise( nNowGear, calcVana, fCarA, fNegrevo, fTe_def);
        if( nRet == NG ){
```

```
            return( false );
        }

        calcVana = p_first->second.fVref_sp;
        bRet = CalcTeMaxSp(nNowGear, (tmpCalcTime / 10.0), tmpVana_sp, calcVana, fNegrevo );
        fCarA = (( calcVana - tmpVana_sp) / (tmpCalcTime / 10.0)) * 1000.0/(60.0*60.0);

        p_tmpCalculate = p_first;
        if( p_tmpCalculate != setCalculateData.begin() ){
            p_tmpCalculate--;
        }
        p_tmpCalculate->second.fCarA = fCarA;

        p_first->second.fVana_sp = calcVana;                                    // Sets analysis speed.
        if( fCarA == 0 ){                                                       // If acceleration is 0
            fNegrevo = tmpNegrevo;                                              // Sets previous engine speed.
        }
        if( fNegrevo < m_fClutch_MeetNe ){
            fNegrevo = m_fClutch_MeetNe;
        }

        fRL = CalcRL( calcVana );                                               // Rolling resistance
        nRet = GetTe( nNowGear, calcVana, fCarA, fNegrevo, fTe);                // Engine torque
        if( nRet == NG ){
            return( false );
        }

        fTeMax = GetLineReviseMaxTorque(fNegrevo);
        if( fTe_def > fTeMax ){
            if( nNowGear >= m_nInitGear ){
                nNowGear = 1;
                p_first = p_start;
                goto CHECK_AGAIN;
            }
        }

        p_first->second.fNegrevo = fNegrevo;                                    // Sets engine speed.
        p_first->second.fRL      = fRL;                                         // Sets rolling resistance.
        p_first->second.fTe      = fTe;                                         // Sets engine torque.
        p_first->second.nGear    = nNowGear;                                    // Sets gear position.
        p_first->second.nFlag    = 1;
        p_first->second.nGearTime = (int)tmpGearTime + (int)tmpCalcTime;
        p_second = p_first; p_second++;                                         // Sets next position.
        if( fCarV < tmpVana_sp ){                                               // If continuous engine speed < analysis speed
            p_first->second.bIdle = true;
            break;
        }
        if( p_first == setCalculateData.end() ){
            break;
        }
        tmpVana_sp = p_first->second.fVana_sp;                                  // Previous analysis speed
        fBeforeA   = p_first->second.fCarA;                                     // Previous acceleration
        tmpNegrevo = p_first->second.fNegrevo;                                  // Sets previous engine speed.
        tmpGearTime = p_first->second.nGearTime;                                // Sets previous gear time.
        tmpCalcTime = p_first->second.nCalcTime;                                // Sets previous required time.

        // ----------------------------------------
        // If 5% normalized engine speed reached
        // ----------------------------------------
        if( fNegrevo > m_fClutch_MeetNe ){
            break;
        }
    }

    return( true );
}
/**/
/*****************************************************************
 * Function name          : Calculate_T3Set
 * Function summary       : Acceleration T3 section setup processing
 * Explanation            : Detailed settings are made for acceleration.
 *                    :
 * Argument (input)       : p_first  : First pointer
 * Argument (input)       : p_second : Next setting pointer
 * Argument (output)      : None
 * Argument (I/O)         : None
 * Return value           : true : Normal    false : Failure
```

```
 * Created by              :
 * Updated on (created on) :
 * Remarks                 :
 ********************************************************************/
bool   TCalculateProc::Calculate_T3Set(map<double,stCalculateData>::iterator p_first,
                    map<double,stCalculateData>::iterator &p_second )
{
    map<double,stCalculateData>::iterator p_tmpCalculate;                    // Temporary pointer
    map<double,stCalculateData>::iterator p_Set;                             // Temporary repeat data
    map<double,stCalculateData>::iterator p_Next;                            // Temporary repeat data (next pointer)
    map<double,stCalculateData>::iterator p_betwn;                           // Identifies pointers before and after.
    map<double,stCalculateData>::iterator p_before;                          // Previous data
    stExceedForce tmpExceedForce;                                            // Structure for search
    stCalculateData tmpCalculateData;                                        // Temporary analysis data
    bool    bRet;
    int     tmpNowGear;                                                      // Temporary gear position
    int     tmpGear;                                                         // Prospective gear position
    double  tmpVref_sp;                                                      // Previous reference speed
    double  tmpVana_sp;                                                      // Previous analysis speed
    double  tmpGearTime;                                                     // Gear required time
    double  tmpCalcTime;                                                     // Required time
    double  fBeforeA;                                                        // Previous acceleration
    int     nBefGear;                                                        // Previous gear position
    bool    bShiftUpFlag;                                                    // Flag to determine shift-up
    bool    bIdleUpFlag;
    int     nRet;                                                            // Return value of function to be recalled
    double  calcVana;                                                        // Analysis speed for calculation
    double  fNeMinLimit;                                                     // Engine speed determination lower limit
    double  fNeMaxTopLimit;                                                  // Max. engine speed rate
    double  tmpV;                                                            // Temporary speed
    double  fTe, fTeMax;                                                     // Torque for calculation
    double  tmpNegrevo;                                                      // Previous engine speed

    p_tmpCalculate = p_first;
    p_tmpCalculate--;                                                        // Sets preceding data point.

    tmpNowGear = p_tmpCalculate->second.nGear;                               // Sets preceding gear position.
    tmpVana_sp = p_tmpCalculate->second.fVana_sp;                            // Previous analysis speed
    tmpVref_sp = p_tmpCalculate->second.fVref_sp;                            // Previous reference vehicle speed
    fBeforeA   = p_tmpCalculate->second.fCarA;                               // Previous reference acceleration
    tmpGearTime= p_tmpCalculate->second.nGearTime;                           // Sets gear hold time.
    tmpCalcTime= p_tmpCalculate->second.nCalcTime;                           // Sets required time.

    bShiftUpFlag = false;
    bIdleUpFlag = false;
    // ----------------------------------
    // Execute processing for starting from intermediate point.
    // ----------------------------------
    if(( p_tmpCalculate->second.bIdle == true )&&
      ( tmpNowGear == 0 )){                                                  // If gear is at neutral
       // Search for prospective gear for conditions.
       memset( &tmpCalculateData, 0x00, sizeof( tmpCalculateData ));         // Initializes temporary data.
       tmpCalculateData.fTimes = p_first->second.fTimes +m_fTg* 10.0;        // Sets gear hold time for shift-up.

       p_Set = p_first;
       p_Next = setCalculateData.find( tmpCalculateData.fTimes );            // Up to end of gear hold time
       tmpGear = 0;                                                          // Initializes prospective gear.
       tmpNowGear = m_nInitGear;
       bRet = Calculate_GearUp( p_Set, p_Next, tmpNowGear );                 // Gear hold check
       bIdleUpFlag = true;
    }else if(( p_tmpCalculate->second.fTe < 0 )&&( tmpNowGear != 0 )){
       // Search for prospective gear for conditions.
       memset( &tmpCalculateData, 0x00, sizeof( tmpCalculateData ));         // Initializes temporary data.
       tmpCalculateData.fTimes = p_first->second.fTimes +m_fTg* 10.0;        // Sets gear hold time for shift-up.

       p_Set = p_first;
       p_Next = setCalculateData.find( tmpCalculateData.fTimes );            // Up to end of gear hold time
       tmpGear = 0;                                                          // Initializes prospective gear.
       nBefGear = tmpNowGear;
       for( tmpNowGear = nBefGear;
            tmpNowGear >= m_nInitGear; tmpNowGear-- ){
          bRet = Calculate_T3Check( p_Set, p_Next, tmpNowGear, nBefGear );   // Gear hold check
          if( bRet == true ){                                                // If gear can be set
             tmpGear = tmpNowGear;                                           // Sets as prospective gear.
             break;
          }
       }
```

```
    if( tmpGear != 0 ){                                         // If prospective gear is set
        tmpNowGear = tmpGear;                                   // Sets the gear.
    }
    if( tmpNowGear <= m_nInitGear ){                            // If no applicable gear is found
        tmpNowGear = m_nInitGear;                               // Forcibly sets gear.
    }
    if( nBefGear != tmpNowGear ){
        bIdleUpFlag = true;
        tmpGearTime = 0;
    }
}else{
    tmpNowGear = p_tmpCalculate->second.nGear;
}


// ------------------------------------------------
// Temporarily apply current gear to all processing steps.
// ------------------------------------------------
for(p_tmpCalculate = p_first;
            p_tmpCalculate != p_second; p_tmpCalculate++ ){
    p_tmpCalculate->second.nGear = tmpNowGear;                  // Sets current gear.
    if( bIdleUpFlag == true ){
        p_tmpCalculate->second.nGearTime = 0;
    }
}

// ---------------------------------
// Determine engine speed availability with current gear.
// ---------------------------------
for( p_tmpCalculate = p_first;
            p_tmpCalculate != p_second;
                p_tmpCalculate++){                              // All sections subject to determination
    if( p_tmpCalculate->second.nGear == 0 ){                    // Check in case of gear being activated
        continue;
    }
    if( p_tmpCalculate != p_first ){
        p_before = p_tmpCalculate;
        p_before--;                                             // Previous set value
        tmpGearTime = p_before->second.nGearTime;               // Previous gear time
        tmpCalcTime = p_before->second.nCalcTime;               // Previous required time
        fBeforeA    = p_before->second.fCarA;                   // Previous acceleration
        tmpVana_sp  = p_before->second.fVana_sp;                // Previous analysis speed
        tmpVref_sp  = p_before->second.fVref_sp;                // Previous reference vehicle speed
        tmpNowGear = p_before->second.nGear;                    // Sets preceding gear.
    }

    if( tmpNowGear == 0 ){                                      // Since 0 gear is not available.
        tmpNowGear = p_tmpCalculate->second.nGear;              // sets current gear.
    }
    nBefGear = tmpNowGear;

    // Set analysis speed (temporarily).
    p_tmpCalculate->second.fVana_sp = p_tmpCalculate->second.fVref_sp;

    p_before = p_tmpCalculate;
    p_before--;                                                 // Sets preceding data point.

    tmpV = p_before->second.fVana_sp;
    tmpCalcTime = p_before->second.nCalcTime;                   // Previous required time

    // ---------------------------------------
    // Obtain acceleration for reaching this-time vehicle speed.
    // ---------------------------------------
    calcVana = p_tmpCalculate->second.fVref_sp;
    bRet = CalcTeMaxSp(tmpNowGear, (tmpCalcTime / 10.0), tmpV, calcVana, tmpNegrevo );
    if( bRet == false ){
        return( false );
    }
    fBeforeA = (( calcVana - tmpV) / (tmpCalcTime / 10.0)) * 1000.0/(60.0*60.0);
    p_before->second.fCarA = fBeforeA;
    p_tmpCalculate->second.fVana_sp = calcVana;


    if(((tmpGearTime + tmpCalcTime) / 10.0) < m_fTg){           // Checks gear hold time.
        bShiftUpFlag = false;
    }else{
        // evolution arithmetic determination of lower limit
```

```
memset( &tmpExceedForce, 0x00, sizeof( tmpExceedForce ));                    // Initializes search structure.
tmpExceedForce.nGear = tmpNowGear;                                           // Sets gear.
fNeMinLimit = m_ExceedForce.find( tmpExceedForce )->fMinNe;                  // Sets lower-limit engine speed.
fNeMaxTopLimit = m_fOutputRotation;

while(1){
   tmpGear = tmpNowGear;                                                     // Holds data before gear setting.
   // Shift-up if maximum  engine speed criterion is exceeded.
   if( tmpNegrevo >= fNeMaxTopLimit){                                        // If engine speed is equal to or more than maximum  engine speed value
      memset( &tmpCalculateData, 0x00, sizeof( tmpCalculateData ));          // Initializes temporary data.
      tmpCalculateData.fTimes = p_tmpCalculate->second.fTimes +m_fTg* 10.0;  // Sets free-running time for shift-up.
      p_Set = p_tmpCalculate;
      p_Next = setCalculateData.find( tmpCalculateData.fTimes );             // Up to end of gear hold time

      tmpGear = tmpNowGear;                                                  // Holds data before gear setting.
      //---------------------------------//
      // Check with gear excess torque ratio.
      //---------------------------------//
      bRet = Calculate_GearUp( p_Set, p_Next, tmpNowGear );                  // Checks until shift-up is impossible with available gear.
      if(( tmpNowGear == tmpGear )&&( tmpNowGear +1 <= m_nMaxGear )){
         tmpNowGear = tmpNowGear +1;
      }
   }

   calcVana = p_tmpCalculate->second.fVref_sp;
   fBeforeA = (( calcVana - tmpV) / (tmpCalcTime / 10.0 )) * 1000.0/(60.0*60.0);
   nRet = GetNe( tmpNowGear,calcVana,tmpNegrevo);                            // Obtains engine speed.
   if( nRet == NG ){
      return(false);
   }
   nRet = GetTe_NotRevise( tmpNowGear, calcVana, fBeforeA, tmpNegrevo, fTe);  // Engine torque (without complement)
   fTeMax = GetLineReviseMaxTorque(tmpNegrevo);                              // Linear interpolation
   if( fTe > fTeMax ){                                                       // Current status maintenance, shift-down
      if( tmpNowGear -1 >= m_nInitGear ){
         Calculate_NeGear( p_tmpCalculate, tmpNowGear );
      }
      break;
   }
   if( tmpNegrevo <= fNeMinLimit){                                           // In case of min. engine speed or less
      if( tmpNowGear -1 >= m_nInitGear ){
         Calculate_NeGear( p_tmpCalculate, tmpNowGear );
      }
   }
   break;
}

if( nBefGear != tmpNowGear ){
   //----------------------------------------
   // Obtain acceleration for reaching this-time vehicle speed.
   //----------------------------------------
   calcVana = p_tmpCalculate->second.fVref_sp;
   bRet = CalcTeMaxSp(tmpNowGear, (tmpCalcTime / 10.0), tmpV, calcVana, tmpNegrevo );
   if( bRet == false ){
      return( false );
   }
   fBeforeA = (( calcVana - tmpV) / (tmpCalcTime / 10.0)) * 1000.0/(60.0*60.0);
   p_before->second.fCarA = fBeforeA;
   p_tmpCalculate->second.fVana_sp = calcVana;
   p_tmpCalculate->second.nGear = tmpNowGear;
}else{
   tmpGear = tmpNowGear;
   calcVana = p_tmpCalculate->second.fVana_sp;
   nRet = GetNe( tmpGear,calcVana,tmpNegrevo);                               // Obtains engine speed.
   if( nRet == NG ){
      return(false);
   }
   nRet = GetTe_NotRevise( tmpGear, calcVana, fBeforeA, tmpNegrevo, fTe);    // Engine torque (without complement)
   fTeMax = GetLineReviseMaxTorque(tmpNegrevo);                              // Linear interpolation
   if( fTe <= fTeMax ){
      memset( &tmpCalculateData, 0x00, sizeof( tmpCalculateData ));          // Initializes temporary data.
      tmpCalculateData.fTimes = p_tmpCalculate->second.fTimes +m_fTg* 10.0;  // Sets free-running time for shift-up.
      p_Set = p_tmpCalculate;
      p_Next = setCalculateData.find( tmpCalculateData.fTimes );             // Up to end of gear hold time
      //---------------------------------//
      // Check with gear excess torque ratio.
      //---------------------------------//
      bRet = Calculate_GearUp( p_Set, p_Next, tmpNowGear );                  // Checks until shift-up is impossible with available gear.
```

```
            }

            // ----------------------------------------
            // Obtain acceleration for reaching this-time vehicle speed.
            // ----------------------------------------
            calcVana = p_tmpCalculate->second.fVref_sp;
            bRet = CalcTeMaxSp(tmpNowGear, (tmpCalcTime / 10.0), tmpV, calcVana, tmpNegrevo );
            if( bRet == false ){
               return( false );
            }
            fBeforeA = (( calcVana - tmpV) / (tmpCalcTime / 10.0)) * 1000.0/(60.0*60.0);
            p_before->second.fCarA = fBeforeA;
            p_tmpCalculate->second.fVana_sp = calcVana;
            p_tmpCalculate->second.nGear = tmpNowGear;
         }
         bShiftUpFlag = true;
      }

      // ------------------------------------
      // Make settings for shift-up T3 section.
      // ------------------------------------
      if(( bShiftUpFlag == true )&&( nBefGear != tmpNowGear )){                    // If shift-up is executed
         bIdleUpFlag = false;
         p_tmpCalculate->second.nGearTime = 0;                                    // Sets gear time to 0.
         p_tmpCalculate->second.nTimeFlg = 3;                                     // Section is t3.
         p_Set = p_tmpCalculate;


         tmpGearTime = 0;
         for( ; p_Set != p_second; p_Set++ ){                                     // Sets remaining gears as set gear.
            p_Set->second.nGear = tmpNowGear;                                     // Sets gear again.
            if( p_Set != p_tmpCalculate ){
               p_Set->second.nGearTime = (int)tmpGearTime + (int)tmpCalcTime;     // Sets gear hold time.
               tmpGearTime = p_Set->second.nGearTime;                             // Previous gear time
            }
            tmpCalcTime = p_Set->second.nCalcTime;                                // Previous required time
         }
         // Write to next data as well.
         if( p_Set != setCalculateData.end() ){
            p_Set->second.nGear = tmpNowGear;                                     // Sets gear again.
            p_Set->second.nGearTime = (int)tmpGearTime + (int)tmpCalcTime;        // Sets gear hold time.
            tmpCalcTime = p_Set->second.nCalcTime;                                // Previous required time
            tmpGearTime = p_Set->second.nGearTime;                                // Previous gear time
         }
      }else{
         if( bIdleUpFlag == true ){
            bIdleUpFlag = false;
            if( tmpGearTime != 0.0 ){
               p_tmpCalculate->second.nGearTime = (int)tmpGearTime +
                                    (int)tmpCalcTime;                             // Sets gear hold time.
            }
         }else{
            p_tmpCalculate->second.nGearTime = (int)tmpGearTime +
                                    (int)tmpCalcTime;                             // Sets gear hold time.
         }
      }

   }

   // ---------------------------------
   // Calculate acceleration again.
   // ---------------------------------
   BtwnCarASet(p_first, p_second);

   return( true );
}
/**/
/*****************************************************************
 * Function name          : Calculate_T3Check
 * Function summary       : Acceleration T3 section confirmation processing
 * Explanation            : Availability of running with applicable gear during acceleration
 *                        : is checked.
 * Argument (input)       : p_first  : First pointer
 * Argument (input)       : p_second : Next setting pointer
 * Argument (input)       : tmpGear  : Gear position
 * Argument (input)       : OrgGear  : Now Gear position
 * Argument (I/O)         : None
```

```
 * Return value            : true : Maintaining OK    false : Maintaining NG
 * Created by              :
 * Updated on (created on) :
 * Remarks                 :
 ********************************************************************/
bool   TCalculateProc::Calculate_T3Check(map<double,stCalculateData>::iterator p_first,
                    map<double,stCalculateData>::iterator p_second,
                    int tmpGear, int OrgGear )
{
    map<double,stCalculateData>::iterator p_tmpCalculate;                          // Temporary pointer
    map<double,stCalculateData>::iterator p_befCalculate;                          // Temporary pointer
    int     nRet;                                                          // Return value of function to be recalled
    bool    bRet;
    bool    bFlag;                                                         // Flag to determine whether current status can be maintained
    double  fTe, fTeMax;                                                       // Torque for calculation
    double  tmpNegrevo;                                                        // Previous engine speed
    double  tmpV;                                                          // Temporary speed
    double  tmpVref_sp;                                                        // Temporary reference speed
    double  tmpCalcTime;
    double  fNeMinLimit;                                                       // Engine speed determination lower limit
    double  fNeMaxTopLimit;                                                    // Max. engine speed rate
    double  befNegrevo;
    double  fCarA;                                                         // Acceleration
    double  calcVana;                                                          // Analysis speed for calculation
    stExceedForce tmpExceedForce;                                                  // Structure for search


    bFlag = true;                                                          // Current status maintain state flag

    if( tmpGear > m_nMaxGear ){
        return(false);
    }

    // evolution arithmetic determination of lower limit
    memset( &tmpExceedForce, 0x00, sizeof( tmpExceedForce ));                      // Initializes search structure.
    tmpExceedForce.nGear = tmpGear;                                            // Sets gear.
    fNeMinLimit = m_ExceedForce.find( tmpExceedForce )->fMinNe;                    // Sets lower-limit engine speed.
    fNeMaxTopLimit = m_fOutputRotation;

    p_befCalculate = p_first;
    p_befCalculate--;                                                      // Preceding speed

    tmpV = p_befCalculate->second.fVana_sp;                                    // Preceding analysis speed
    tmpVref_sp = p_befCalculate->second.fVref_sp;                              // Preceding reference speed
    fCarA = p_befCalculate->second.fCarA;                                      // Acceleration
    tmpCalcTime = p_befCalculate->second.nCalcTime;
    befNegrevo = p_befCalculate->second.fNegrevo;

    // ----------------------------------------
    // Obtain acceleration for reaching this-time vehicle speed.
    // ----------------------------------------
    calcVana = p_first->second.fVref_sp;
    fCarA = (( calcVana - tmpV) / (tmpCalcTime / 10.0)) * 1000.0/(60.0*60.0);

    nRet = GetNe( tmpGear,calcVana,tmpNegrevo);                                // Obtains engine speed.
    if( nRet == NG ){
        return(false);
    }
    // Shift-up if maximum  engine speed criterion is exceeded.
    if(tmpNegrevo >= fNeMaxTopLimit){                                      // If engine speed is equal to or more than maximum  engine speed value
        return(false);
    }
    if( tmpNegrevo <= fNeMinLimit){                                        // In case of min. engine speed or less
        return(false);                                                     // Does not execute shift-up.
    }

    nRet = GetTe_NotRevise( tmpGear, calcVana, fCarA, tmpNegrevo, fTe);            // Engine torque (without complement)
    fTeMax = GetLineReviseMaxTorque(tmpNegrevo);                               // Linear interpolation
    if( fTe > fTeMax ){                                                    // Current status maintenance, shift-down
        if( tmpGear != OrgGear ){
            return(false);
        }
    }

    bRet = CalcTeMaxSp(tmpGear, (tmpCalcTime / 10.0), tmpV, calcVana, tmpNegrevo );
    if( bRet == false ){
        return( false );
```

```
    }
    fCarA = (( calcVana - tmpV) / (tmpCalcTime / 10.0)) * 1000.0/(60.0*60.0);
    p_befCalculate->second.fCarA = fCarA;
    p_first->second.fVana_sp = calcVana;


    if( OrgGear != tmpGear ){
        calcVana = p_first->second.fVref_sp;
        fCarA = (( calcVana - tmpV) / (tmpCalcTime / 10.0)) * 1000.0/(60.0*60.0);
        bRet = CheckForce( tmpGear, calcVana, fCarA );                    // Determines shift-up availability.
        if( bRet != true ){                                               // If shift-up is unavailable
            return(false);
        }
    }

    // ----------------------------
    // Check for gear hold time
    // ----------------------------
    for( p_tmpCalculate = p_first;
                p_tmpCalculate != p_second;
                    p_tmpCalculate++){                                    // All sections subject to determination
        p_befCalculate = p_tmpCalculate;
        p_befCalculate--;                                                 // Preceding speed

        tmpV = p_befCalculate->second.fVana_sp;                           // Preceding analysis speed
        calcVana = p_tmpCalculate->second.fVref_sp;
        tmpCalcTime = p_befCalculate->second.nCalcTime;
        bRet = CalcTeMaxSp(tmpGear, (tmpCalcTime / 10.0), tmpV, calcVana, tmpNegrevo );
        if( bRet == false ){
            return( false );
        }
        fCarA = (( calcVana - tmpV) / (tmpCalcTime / 10.0)) * 1000.0/(60.0*60.0);
        p_tmpCalculate->second.fVana_sp = calcVana;
        p_befCalculate->second.fCarA = fCarA;
        if( fCarA < 0 ){
            break;
        }

        // Shift-up if maximum  engine speed criterion is exceeded.
        if(tmpNegrevo >= fNeMaxTopLimit){                                 // If engine speed is equal to or more than maximum  engine speed value
            return(false);                                                // Executes shift-up. (Enables maintaining current status and sets gear to top.))
        }
        if( tmpNegrevo <= fNeMinLimit){                                   // In case of min. engine speed or less
            return(false);                                                // Does not execute shift-up.
        }

        if( OrgGear != tmpGear ){
            calcVana = p_tmpCalculate->second.fVref_sp;
            fCarA = (( calcVana - tmpV) / (tmpCalcTime / 10.0)) * 1000.0/(60.0*60.0);
            nRet = GetTe_NotRevise( tmpGear, calcVana, fCarA, tmpNegrevo, fTe);       // Engine torque (without complement)
            fTeMax = GetLineReviseMaxTorque(tmpNegrevo);                  // Linear interpolation
            if( fTe > fTeMax ){                                           // If max. torque is exceeded
                return(false);
            }
        }
    }

    if( bFlag == true ){                                                  // If current status can be maintained by holding specified gear
        bRet = true;
    }else{
        bRet = false;
    }

    return(bRet);

}
/**/
/*****************************************************************
 * Function name          : Calculate_GearUp
 * Function summary       : Acceleration T3 section confirmation processing
 * Explanation            : Availability of running with applicable gear during acceleration
 *                        : is checked.
 * Argument (input)       : p_first  : First pointer
 * Argument (input)       : p_second : Next setting pointer
 * Argument (I/O)         : tmpGear  : Gear position
 * Return value           : true : Maintaining OK    false : Maintaining NG
 * Created by             :
```

```
 * Updated on (created on) :
 * Remarks                 :
 ********************************************************************/
bool   TCalculateProc::Calculate_GearUp(map<double,stCalculateData>::iterator p_first,
                    map<double,stCalculateData>::iterator p_second,
                    int &tmpGear )
{
    map<double,stCalculateData>::iterator p_tmpCalculate;                    // Temporary pointer
    map<double,stCalculateData>::iterator p_befCalculate;                    // Temporary pointer
    map<int,double> mDiffSpeed;                                        // diffrent reference speed data
    map<int,double>::iterator p_DiffSpeed;                                  // diffrent reference spped data pointer
    int     nRet;                                                   // Return value of function to be recalled
    bool    bRet;
    bool    bFlag;                                                  // Flag to determine whether current status can be maintained
    int     OrgGear;                                                // Setting before gear potition
    int     nSetGear;
    double  fDiffSpeed;
    double  fTe, fTeMax;                                               // Torque for calculation
    double  tmpNegrevo;                                                // Previous engine speed
    double  tmpV;                                                   // Temporary speed
    double  tmpVref_sp;                                                // Temporary reference speed
    double  tmpCalcTime;
    double  fNeMinLimit;                                               // Engine speed determination lower limit
    double  fNeMaxTopLimit;                                               // Max. engine speed rate
    double  befNegrevo;
    double  fCarA;                                                  // Acceleration
    double  calcVana;                                                  // Analysis speed for calculation
    stExceedForce tmpExceedForce;                                             // Structure for search

    OrgGear = tmpGear;
    bFlag = true;                                                   // Current status maintain state flag

    if( tmpGear > m_nMaxGear ){
        return(false);
    }

    // ----------------------------------//
    // Check with gear excess torque ratio.
    // ----------------------------------//
    if( tmpGear + 3 > m_nMaxGear ){
        tmpGear = m_nMaxGear;
    }else{
        tmpGear = tmpGear +3;
    }

    for(;tmpGear >= OrgGear ; tmpGear-- ){                                  // Searches for optimum gear in available range.
        // evolution arithmetic determination of lower limit
        memset( &tmpExceedForce, 0x00, sizeof( tmpExceedForce ));                    // Initializes search structure.
        tmpExceedForce.nGear = tmpGear;                                     // Sets gear.
        fNeMinLimit = m_ExceedForce.find( tmpExceedForce )->fMinNe;                  // Sets lower-limit engine speed.
        fNeMaxTopLimit = m_fOutputRotation;

        p_befCalculate = p_first;
        p_befCalculate--;                                               // Preceding speed

        tmpV = p_befCalculate->second.fVana_sp;                                 // Preceding analysis speed
        tmpVref_sp = p_befCalculate->second.fVref_sp;                           // Preceding reference speed
        fCarA = p_befCalculate->second.fCarA;                                   // Acceleration
        tmpCalcTime = p_befCalculate->second.nCalcTime;
        befNegrevo = p_befCalculate->second.fNegrevo;

        // ---------------------------------------
        // Obtain acceleration for reaching this-time vehicle speed.
        // ---------------------------------------
        calcVana = p_first->second.fVref_sp;
        fCarA = (( calcVana - tmpV) / (tmpCalcTime / 10.0)) * 1000.0/(60.0*60.0);

        nRet = GetNe( tmpGear,calcVana,tmpNegrevo);                             // Obtains engine speed.
        if( nRet == NG ){
            return(false);
        }
        // Shift-up if maximum  engine speed criterion is exceeded.
        if(tmpNegrevo >= fNeMaxTopLimit){                                   // If engine speed is equal to or more than maximum  engine speed value
            continue;
        }
        if( tmpNegrevo <= fNeMinLimit){                                     // In case of min. engine speed or less
            continue;                                                   // Does not execute shift-up.
```

```
    }

    nRet = GetTe_NotRevise( tmpGear, calcVana, fCarA, tmpNegrevo, fTe);          // Engine torque (without complement)
    fTeMax = GetLineReviseMaxTorque(tmpNegrevo);                                 // Linear interpolation
    if( fTe > fTeMax ){                                                          // Current status maintenance, shift-down
      if( tmpGear != OrgGear ){
        continue;
      }
    }

    bRet = CalcTeMaxSp(tmpGear, (tmpCalcTime / 10.0), tmpV, calcVana, tmpNegrevo );
    if( bRet == false ){
      return( false );
    }
    fCarA = (( calcVana - tmpV) / (tmpCalcTime / 10.0)) * 1000.0/(60.0*60.0);
    p_befCalculate->second.fCarA = fCarA;
    p_first->second.fVana_sp = calcVana;

    if( OrgGear != tmpGear ){
      calcVana = p_first->second.fVref_sp;
      fCarA = (( calcVana - tmpV) / (tmpCalcTime / 10.0)) * 1000.0/(60.0*60.0);
      bRet = CheckForce( tmpGear, calcVana, fCarA );                             // Determines shift-up availability.
      if( bRet != true ){                                                        // If shift-up is unavailable
        continue;
      }
    }

    // ----------------------------
    // Check for gear hold time
    // ----------------------------
    nSetGear = tmpGear;
    fDiffSpeed = 0;
    for( p_tmpCalculate = p_first;
           p_tmpCalculate != p_second;
             p_tmpCalculate++){                                                  // All sections subject to determination
      p_befCalculate = p_tmpCalculate;
      p_befCalculate--;                                                          // Preceding speed

      tmpV = p_befCalculate->second.fVana_sp;                                    // Preceding analysis speed
      calcVana = p_tmpCalculate->second.fVref_sp;
      tmpCalcTime = p_befCalculate->second.nCalcTime;
      bRet = CalcTeMaxSp(tmpGear, (tmpCalcTime / 10.0), tmpV, calcVana, tmpNegrevo );
      if( bRet == false ){
        return( false );
      }
      fCarA = (( calcVana - tmpV) / (tmpCalcTime / 10.0)) * 1000.0/(60.0*60.0);
      p_tmpCalculate->second.fVana_sp = calcVana;
      p_befCalculate->second.fCarA = fCarA;
      if( fCarA < 0 ){
        break;
      }

      // Shift-up if maximum  engine speed criterion is exceeded.
      if(tmpNegrevo >= fNeMaxTopLimit){                                          // If engine speed is equal to or more than maximum  engine speed value
        nSetGear = 0;                                                            // Executes shift-up. (Enables maintaining current status and sets gear to top.))
        break;
      }
      if( tmpNegrevo <= fNeMinLimit){                                            // In case of min. engine speed or less
        nSetGear = 0;                                                            // Does not execute shift-up.
        break;
      }
      fDiffSpeed = fDiffSpeed +
             (p_tmpCalculate->second.fVref_sp -
              p_tmpCalculate->second.fVana_sp );
    }

    if( nSetGear != 0 ){
      mDiffSpeed.insert(pair<int,double>(nSetGear, fDiffSpeed) );                // Sets analysis time in pattern information
    }
  }

  if( mDiffSpeed.empty() != true ){
    p_DiffSpeed = mDiffSpeed.begin();
    fDiffSpeed = p_DiffSpeed->second;
    nSetGear = p_DiffSpeed->first;
    for( p_DiffSpeed = mDiffSpeed.begin();
           p_DiffSpeed != mDiffSpeed.end();
```

```
                    p_DiffSpeed++ ){
            if( fabs( p_DiffSpeed->second ) <= fabs( fDiffSpeed ) ){
                if( nSetGear < p_DiffSpeed->first ){
                    fDiffSpeed = p_DiffSpeed->second;
                    nSetGear = p_DiffSpeed->first;
                }
            }
        }
        mDiffSpeed.erase( mDiffSpeed.begin(), mDiffSpeed.end() );
        mDiffSpeed.clear();
    }else{
        nSetGear = 0;
    }

    if( nSetGear == 0 ){
        tmpGear = OrgGear;
        bRet = false;
    }else{
        tmpGear = nSetGear;
        bRet = true;
    }

    return(bRet);

}
/**/
/*****************************************************************
 * Function name          : Calculate_NeGear
 * Function summary       : Min. normal engine speed gear setting
 * Explanation            : Until min. normal engine speed (min. engine speed) is reached
 *                        : gear is shifted down.
 * Argument (input)       : p_first : First pointer
 * Argument (output)      : None
 * Argument (I/O)         : nGear : Gear position
 * Return value           :
 * Created by             :
 * Updated on (created on) :
 * Remarks                :
 *****************************************************************/
bool  TCalculateProc::Calculate_NeGear(map<double,stCalculateData>::iterator p_first, int &nGear )
{
    map<double,stCalculateData>::iterator p_befCalculate;                    // Temporary pointer
    map<double,stCalculateData>::iterator p_tmpCalculate;                    // Temporary pointer
    map<double,stCalculateData>::iterator p_second;
    map<int,double> mDiffSpeed;                                              // diffrent reference speed data
    map<int,double>::iterator p_DiffSpeed;                                   // diffrent reference spped data pointer
    stExceedForce tmpExceedForce;                                            // Structure for search
    stCalculateData tmpCalculateData;                                        // Temporary analysis data
    int     nRet;
    bool    bRet;
    double  fNeMinLimit;                                                     // Engine speed determination lower limit
    double  tmpNegrevo;                                                      // Previous engine speed
    double  tmpV;                                                            // Temporary speed
    double  tmpVref_sp;                                                      // Temporary reference speed
    double  fCarA;                                                           // Acceleration data for calculation result
    double  calcVana;                                                        // Speed data for calculation
    double  tmpCalcTime;
    double  fDiffSpeed;
    int     nOrgGear;
    int     nSetGear;

    if( nGear < m_nInitGear ){
        nGear = m_nInitGear;
    }
    nOrgGear = nGear;
    nSetGear = 0;

    p_befCalculate = p_first;
    p_befCalculate--;                                                        // Preceding speed

    tmpV = p_befCalculate->second.fVana_sp;                                  // Preceding analysis speed
    tmpVref_sp = p_befCalculate->second.fVref_sp;                            // Preceding reference speed
    fCarA = p_befCalculate->second.fCarA;                                    // Acceleration
    tmpCalcTime = p_befCalculate->second.nCalcTime;
    calcVana = tmpV + fCarA * tmpCalcTime /10.0 * 3.6;                       // Analysis vehicle speed

    tmpCalculateData.fTimes = p_first->second.fTimes + m_fTg* 10.0;          // End of gear keep point
```

```
p_second = setCalculateData.find( tmpCalculateData.fTimes );

for( ; nGear >= m_nInitGear; nGear-- ){
    memset( &tmpExceedForce, 0x00, sizeof( tmpExceedForce ));                    // Initializes search structure.
    tmpExceedForce.nGear = nGear;                                                // Sets gear.
    fNeMinLimit = m_ExceedForce.find( tmpExceedForce )->fMinNe;                  // Sets lower-limit engine speed.

    calcVana = p_first->second.fVref_sp;
    bRet = CalcTeMaxSp(nGear, (tmpCalcTime / 10.0), tmpV, calcVana, tmpNegrevo );
    if( bRet == false ){
        return( false );
    }
    fCarA = (( calcVana - tmpV) / (tmpCalcTime / 10.0)) * 1000.0/(60.0*60.0);
    p_befCalculate->second.fCarA = fCarA;
    p_first->second.fVana_sp = calcVana;

    if( tmpNegrevo > fNeMinLimit){                                               // In case of more than min. engine speed
        calcVana = p_first->second.fVref_sp;
        fCarA = (( calcVana - tmpV) / (tmpCalcTime / 10.0)) * 1000.0/(60.0*60.0);
        nRet = GetNe( nGear,calcVana,tmpNegrevo);                                // Obtains engine speed.
        if( nRet != OK ){
            break;
        }

        nSetGear = nGear;
        fDiffSpeed = 0.0;

        for( p_tmpCalculate = p_first;
                        p_tmpCalculate != p_second;
                        p_tmpCalculate++){                                       // All sections subject to determination
            p_befCalculate = p_tmpCalculate;
            p_befCalculate--;                                                    // Preceding speed

            tmpV = p_befCalculate->second.fVana_sp;                              // Preceding analysis speed
            calcVana = p_tmpCalculate->second.fVref_sp;
            tmpCalcTime = p_befCalculate->second.nCalcTime;
            bRet = CalcTeMaxSp(nSetGear, (tmpCalcTime / 10.0), tmpV, calcVana, tmpNegrevo );
            if( bRet == false ){
                return( false );
            }
            fCarA = (( calcVana - tmpV) / (tmpCalcTime / 10.0)) * 1000.0/(60.0*60.0);
            p_tmpCalculate->second.fVana_sp = calcVana;
            p_befCalculate->second.fCarA = fCarA;
            if( fCarA < 0 ){
                if( p_tmpCalculate == p_first ){
                    nSetGear = 0;
                }
                break;
            }

            if( nOrgGear != nSetGear ){
                // Shift-up if maximum  engine speed criterion is exceeded.
                if(tmpNegrevo >= m_fOutputRotation){                             // If engine speed is equal to or more than maximum  engine speed value
                    nSetGear = 0;
                    break;
                }
            }
            fDiffSpeed = fDiffSpeed +
                    (p_tmpCalculate->second.fVref_sp -
                     p_tmpCalculate->second.fVana_sp );
        }
        if( nSetGear != 0 ){
            mDiffSpeed.insert(pair<int,double>(nSetGear, fDiffSpeed) );          // Sets analysis time in pattern information
        }
    }
}

if( mDiffSpeed.empty() != true ){
    p_DiffSpeed = mDiffSpeed.begin();
    fDiffSpeed = p_DiffSpeed->second;
    nSetGear = p_DiffSpeed->first;
    for( p_DiffSpeed = mDiffSpeed.begin();
                     p_DiffSpeed != mDiffSpeed.end();
                     p_DiffSpeed++ ){
        if( fabs( p_DiffSpeed->second ) <= fabs( fDiffSpeed ) ){
            if( nSetGear < p_DiffSpeed->first ){
                fDiffSpeed = p_DiffSpeed->second;
```

```
                nSetGear = p_DiffSpeed->first;
              }
            }
        }
        mDiffSpeed.erase( mDiffSpeed.begin(), mDiffSpeed.end() );
        mDiffSpeed.clear();
    }else{
        nSetGear = 0;
    }

    if( nSetGear == 0 ){
        nGear = nOrgGear;
    }else{
        nGear = nSetGear;
    }

    return(true);
}
/**/
/*****************************************************************
 * Function name          : Calculate_T6Set
 * Function summary       : Deceleration T6 section setup processing
 * Explanation            : Detailed settings are made for deceleration.
 *                        :
 * Argument (input)       : p_first  : First pointer
 * Argument (input)       : p_second : Next setting pointer
 * Argument (output)      : None
 * Argument (I/O)         : None
 * Return value           : true : Normal    false : Failure
 * Created by             :
 * Updated on (created on) :
 * Remarks                :
 *****************************************************************/
bool   TCalculateProc::Calculate_T6Set(map<double,stCalculateData>::iterator p_first,
                    map<double,stCalculateData>::iterator p_second )
{
    map<double,stCalculateData>::iterator p_tmpCalculate;                    // Temporary pointer
    map<double,stCalculateData>::iterator p_Set;                             // Temporary repeat data
    map<double,stCalculateData>::iterator p_Next;                            // Temporary repeat data (next pointer)
    map<double,stCalculateData>::iterator p_betwn;                           // Identifies pointers before and after.
    map<double,stCalculateData>::iterator p_before;                          // Previous data
    int     nRet;                                                    // Return value of function to be recalled
    int     tmpNowGear;                                              // Temporary gear
    double  tmpVana_sp;                                              // Previous analysis speed
    double  tmpNegrevo;                                              // Previous engine speed
    double  tmpGearTime;                                             // Gear required time
    double  tmpCalcTime;                                             // Required time
    double  fBeforeA;                                                // Previous acceleration

    p_tmpCalculate = p_first;
    p_tmpCalculate--;                                                // Sets preceding data point.


    tmpNowGear = p_tmpCalculate->second.nGear;                       // Sets preceding gear.
    tmpVana_sp = p_tmpCalculate->second.fVana_sp;                    // Previous analysis speed
    fBeforeA   = p_tmpCalculate->second.fCarA;                       // Previous reference acceleration
    tmpGearTime= p_tmpCalculate->second.nGearTime;                   // Sets gear hold time.
    tmpCalcTime= p_tmpCalculate->second.nCalcTime;                   // Sets required time.
    // ------------------------------------------------
    // Temporarily apply current gear to all processing steps.
    // ------------------------------------------------
    for(;p_tmpCalculate != p_second; p_tmpCalculate++ ){
        p_tmpCalculate->second.nGear = tmpNowGear;                   // Sets current gear.
    }

    // ---------------------------------
    // Determine engine speed availability with current gear.
    // ---------------------------------
    for( p_tmpCalculate = p_first;
                p_tmpCalculate != p_second;
                    p_tmpCalculate++){                               // All sections subject to determination
        if( p_tmpCalculate->second.nGear == 0 ){                     // Check in case of gear being activated
            continue;
        }

        if( p_tmpCalculate != p_first ){
            p_before = p_tmpCalculate;
```

```
            p_before--;                                                 // Previous set value
            tmpGearTime = p_before->second.nGearTime;                       // Previous gear time
            tmpCalcTime = p_before->second.nCalcTime;                       // Previous required time
            fBeforeA    = p_before->second.fCarA;                       // Previous acceleration
            tmpVana_sp  = p_before->second.fVana_sp;                        // Previous analysis speed
            tmpNowGear = p_before->second.nGear;                            // Sets preceding gear.
        }

        nRet = GetNe( p_tmpCalculate->second.nGear,tmpVana_sp,tmpNegrevo);          // Obtains engine speed.
        if (nRet != OK) return( false );

        // ------------------------------------
        // Make settings for shift-down T6 section.
        // ------------------------------------
        p_tmpCalculate->second.nGearTime = (int)tmpGearTime + (int)tmpCalcTime;     // Sets gear hold time.
        // Set analysis speed (temporarily).
        p_tmpCalculate->second.fVana_sp = tmpVana_sp +
                        fBeforeA * tmpCalcTime /10.0 * 3.6;             // Analysis vehicle speed
    }

    // --------------------------------
    // Calculate acceleration again.
    // --------------------------------
    BtwnCarASet(p_first, p_second);

    return( true );
}
/**/
// ###########################################################################
// ---------------------------------------------------------------------------
// Calculation processing starts here.
// ---------------------------------------------------------------------------
// ###########################################################################
/**/
/*****************************************************************
 * Function name          : CalcRL
 * Function summary       : Rolling resistance calculation processing
 * Explanation            : Rolling resistance is calculated.
 *                        :
 * Argument (input)       : fV : Vehicle speed
 * Argument (output)      : None
 * Argument (I/O)         : None
 * Return value           : double Rolling resistance value
 * Created by             :
 * Updated on (created on) :
 * Remarks                :
 *****************************************************************/
double   TCalculateProc::CalcRL(double fV)
{
    string szData,szKey;
    double fCarM;
    double fRL;
    double fCarB, fCarH;

    // Read and calculate weight data.
    fCarM = GetCarWeight(false);
    fCarB = m_fOverWidth;                                               // Car width
    fCarH = m_fOverHeight;                                              // Car height
    fRL = ((double)((0.00513 + 17.6/fCarM) * fCarM)) +
            ((double)((0.00299 * fCarB * fCarH - 0.000832)) * (fV * fV));
    return( fRL );
}
/**/
/*****************************************************************
 * Function name          : GetCarWeight
 * Function summary       : Curb vehicle weight data calculation processing
 * Explanation            : Curb vehicle weight is calculated.
 *                        :
 * Argument (input)       : bFlag : If true, equivalent rotational inertia mass ratio is included, and not included if false.
 * Argument (output)      : None
 * Argument (I/O)         : None
 * Return value           : double Curb vehicle weight value
 * Created by             :
 * Updated on (created on) :
 * Remarks                :
 *****************************************************************/
double   TCalculateProc::GetCarWeight(bool bFlag, int nGear)
```

```
{
    double fW;
    double fGearHi;

    if (bFlag){
       GetGearHi(nGear,fGearHi);
       fW = m_fCarMe +
            m_fCarMe * m_fMFact +
            m_fCarMe * m_fEFact * fGearHi*fGearHi +
            m_fCarMc +
            m_fPersonW;
    }else{
       fW = m_fCarMc  + m_fCarMe + m_fPersonW;
    }
    return( fW );
}
/**/
/*****************************************************************
 * Function name          : GetGearHi
 * Function summary       : Gear ratio acquisition processing
 * Explanation            : Gear ratio is obtained from gear position.
 *                        :
 * Argument (input)       : nGear   : Gear position
 * Argument (output)      : fGearHi : Gear ratio
 * Argument (I/O)         : None
 * Return value           : OK : Normal   NG : Failure
 * Created by             :
 * Updated on (created on) :
 * Remarks                :
 *****************************************************************/
int   TCalculateProc::GetGearHi(int nGear,double &fGearHi)
{

    if( m_fGearHi.empty() == true ){                                    // If no data exists
       fGearHi = 1;                                                     // Temporarily sets gear ratio to 1.
       return( NG );
    }

    if( nGear > (int)(m_fGearHi.size()) ){                              // In case of larger than entered gear ratio
       fGearHi = 1;                                                     // Temporarily sets gear ratio to 1.
       return( NG );
    }

    fGearHi = m_fGearHi[nGear-1];                                       // Sets gear ratio.

    return( OK );
}
/**/
/*****************************************************************
 * Function name          : GetGeariN
 * Function summary       : Gear ratio acquisition (including final reduction ratio)
 * Explanation            : Gear ratio including final reduction ratio is obtained
 *                        : from gear position.
 * Argument (input)       : nGrear  : Gear position
 * Argument (output)      : fGearti : Gear ratio
 * Argument (I/O)         : None
 * Return value           : OK : Normal   NG : Failure
 * Created by             :
 * Updated on (created on) :
 * Remarks                :
 *****************************************************************/
int   TCalculateProc::GetGeariN(int nGrear,double &fGearti)
{

    int        nRet;
    string     szKey;
    string     sznGearData;                                             // n'th-gear ratio data
    double     fnGearData, fLastReduceGear;

    // n'th-gear ratio
    nRet = GetGearHi(nGrear, fnGearData);
    if( nRet != OK ){
       return( NG );
    }

    fLastReduceGear = m_fLastReduceGear;
    fGearti = fnGearData * fLastReduceGear;
```

```
    return( OK );
}
/**/
/*****************************************************************
 * Function name          : GetNe
 * Function summary       : Engine speed calculation processing
 * Explanation            : Engine speed is calculated.
 *                        :
 * Argument (input)       : nGear : Gear position
 * Argument (input)       : fVg   : Vehicle speed (Vg)
 * Argument (output)      : fNe   : Engine speed
 * Argument (I/O)         : None
 * Return value           : OK : Normal   NG : Failure
 * Created by             :
 * Updated on (created on) :
 * Remarks                :
 *****************************************************************/
int   TCalculateProc::GetNe( int nGear,double fVg,double &fNe)
{
    string szKey,szData;
    int    nRet;
    double fGearti;
    double fTarR;                                                   // Tire rolling radius data

    fTarR = m_fTarR;                                                // Tire rolling radius
    nRet = GetGeariN(nGear,fGearti);
    if (nRet != OK) return( NG );

    fNe = fVg / 60.0 * fGearti * 1000.0 / (2.0 * CalculateProc->m_fPAI*fTarR);
    return( OK );
}
/**/
/*****************************************************************
 * Function name          : GetV
 * Function summary       : Speed calculation processing
 * Explanation            : Speed is calculated.
 *                        :
 * Argument (input)       : nGear : Gear position
 * Argument (input)       : fNe   : Engine speed
 * Argument (output)      : fVg   : Vehicle speed (Vg)
 * Argument (I/O)         : None
 * Return value           : OK : Normal   NG : Failure
 * Created by             :
 * Updated on (created on) :
 * Remarks                :
 *****************************************************************/
int   TCalculateProc::GetV( int nGear,double fNe,double &fVg)
{
    string szKey,szData;
    int    nRet;
    double fGearti;
    double fTarR;                                                   // Tire rolling radius data

    fTarR = m_fTarR;                                                // Tire rolling radius
    nRet = GetGeariN(nGear,fGearti);
    if (nRet != OK) return( NG );

    fVg = fNe * 60.0 / fGearti / 1000.0 * (2.0 * CalculateProc->m_fPAI*fTarR);
    return( OK );
}
/**/
/*****************************************************************
 * Function name          : GetTe
 * Function summary       : Torque calculation processing
 * Explanation            : Torque is calculated. However, interpolation data is
 *                        : considered.
 * Argument (input)       : nGear : Gear position
 * Argument (input)       : fV    : Vehicle speed (fV)
 * Argument (input)       : fA    : Acceleration for fV
 * Argument (input)       : fNe   : Engine speed
 * Argument (output)      : fTe   : Torque
 * Argument (I/O)         : None
 * Return value           : OK : Normal   NG : Failure
 * Created by             :
 * Updated on (created on) :
 * Remarks                :
```

```
 ****************************************************************/
int   TCalculateProc::GetTe( int nGear,double fV,double fA, double fNe, double &fTe)
{
    string szKey,szData;
    int    nRet;
    double fnGearPass;                                          // n'th-grear ratio (transmission efficiency) data
    double fTarR;                                               // Tire rolling radius data
    double fCarMt;                                              // Vehicle body weight
    double fKg ;                                                // Gravitational acceleration
    double fGearti,fRL;
    double fUd;                                                 // Transmission efficiency of final speed reducer
    double tmpTe;

    // Gravitational acceleration
    nRet = GetKG(fKg);
    if (nRet != OK) return( NG );

    // Vehicle body weight
    fCarMt = GetCarWeight(true,nGear);

    // Obtain gear transmission efficiency.
    fnGearPass = GetGearPass(nGear);

    nRet = GetGeariN(nGear,fGearti);
    if(nRet == NG){
        return( NG );
    }

    fRL = CalcRL(fV);
    // Tire rolling radius data

    fTarR = m_fTarR;
    // Final reduction ratio (transmission efficiency)
    fUd = m_fUd;

    // Engine torque  Te = {(Mt * Alfa / g) + RL} * (1000 * rd) / (Gti * Ut)
    fTe = ((fKg*fTarR)/( fGearti * fnGearPass * fUd))*( fRL + (fCarMt / fKg) * fA );

    tmpTe = GetLineReviseMaxTorque(fNe);                        // Linear interpolation
    if(( fNe > 0 )&&( fTe > tmpTe )){                           // If max. torque is exceeded
        if( tmpTe != 0.0 ){
            fTe = tmpTe;
        }
    }

    return( OK );
}
/**/
/*********************************************************************
 * Function name          : GetTe_NotRevise
 * Function summary       : Torque calculation processing (no correction)
 * Explanation            : Torque is calculated.
 *                        :
 * Argument (input)       : nGear : Gear position
 * Argument (input)       : fV    : Vehicle speed (fV)
 * Argument (input)       : fA    : Acceleration for fV
 * Argument (input)       : fNe   : Engine speed
 * Argument (output)      : fTe   : Torque
 * Argument (I/O)         : None
 * Return value           : OK : Normal   NG : Failure
 * Created by             :
 * Updated on (created on) :
 * Remarks                :
 ********************************************************************/
int   TCalculateProc::GetTe_NotRevise( int nGear,double fV,double fA, double fNe, double &fTe)
{
    string szKey,szData;
    int    nRet;
    double fnGearPass;                                          // n'th-grear ratio (transmission efficiency) data
    double fTarR;                                               // Tire rolling radius data
    double fCarMt;                                              // Vehicle body weight
    double fKg ;                                                // Gravitational acceleration
    double fGearti,fRL;
    double fUd;                                                 // Transmission efficiency of final speed reducer

    // Gravitational acceleration
    nRet = GetKG(fKg);
```

```
    if (nRet != OK) return( NG );

    // Vehicle body weight
    fCarMt = GetCarWeight(true,nGear);

    // Obtain gear transmission efficiency.
    fnGearPass = GetGearPass(nGear);

    nRet = GetGeariN(nGear,fGearti);
    if(nRet == NG){
        return( NG );
    }

    fRL = CalcRL(fV);
    // Tire rolling radius data

    fTarR = m_fTarR;
    // Final reduction ratio (transmission efficiency)
    fUd = m_fUd;

    // Engine torque  Te = (Mt * Alfa / g + RL ) * (1000 * rd) / (Gti * Ut)
    fTe = ((fKg*fTarR)/( fGearti * fnGearPass * fUd))*( fRL + (fCarMt / fKg) * fA );

    return( OK );
}
/**/
/*********************************************************************
 * Function name          : GetCalculateDataFileName
 * Function summary       : Output file name acquisition processing
 * Explanation            : Output file name is set.
 *                        :
 * Argument (input)       : None
 * Argument (output)      : szFile : Output file name
 * Argument (I/O)         : None
 * Return value           :
 * Created by             :
 * Updated on (created on) :
 * Remarks                :
 *********************************************************************/
void    TCalculateProc::GetCalculateDataFileName(string & szFile )
{
    string  szName,szExt;
    string  szData;
    string  szCurrentDateTime;


    szFile = m_OutputData;                                          // Sets default output file name.
    if( szFile == "" ){                                             // If no file name is set
        cerr << "Please. Output File=";
        cin >> szFile;
    }

    return;
}
/**/
/*********************************************************************
 * Function name          : DispCalculateData
 * Function summary       : Processing for parameter display during processing
 * Explanation            : Specification data from read file is
 *                        : displayed on screen.
 * Argument (input)       : None
 * Argument (output)      : None
 * Argument (I/O)         : None
 * Return value           :
 * Created by             :
 * Updated on (created on) :
 * Remarks                :
 *********************************************************************/
void    TCalculateProc::DispCalculateData(void)
{
    set<stExceedForce>::iterator  p_ExceedForce;                    // Excess force ratio data
    char buf[256];
    double fCarM;
    double fGVWCarM;
    double fDW;

    fGVWCarM = m_fCarMaxW + m_fCarIniW + (m_fPersonW * m_fPersons);
```

```
    fCarM = GetCarWeight(false);

    sprintf( buf, " GVW   =%8.2f[kg]\n", fGVWCarM );
    cout << buf;
    sprintf( buf, " W0    =%8.2f[kg], Wtest =%8.2f[kg]\n", m_fCarIniW , fCarM );
    cout << buf;
    sprintf( buf, " Width =%8.3f[m],  Height=%8.3f[m], Tire radius=%8.3f[m]\n",
                            m_fOverWidth,
                            m_fOverHeight,
                            m_fTarR );
    cout << buf;
    sprintf( buf, " Crew  =%3d\n", (int)(m_fPersons) );
    cout << buf;
    sprintf( buf, "\n" );
    cout << buf;
    sprintf( buf, " Nidle =%8.2f[rpm], Nrate =%8.2f[rpm], Nex  =%8.2f[rpm]\n",
                            m_fIdleNe,
                            m_fStandardOutputRotation,
                            m_fOutputRotation );
    cout << buf;
    sprintf( buf, " Nes   =%8.2f[rpm], Nec   =%8.2f[rpm]\n",
                            m_fClutch_MeetNe,
                            m_fClutch_ReleaseNe );
    cout << buf;
    sprintf( buf, " MuAir =%10.6f [kgf/(km/h)^2], MuRoll =%10.6f [kgf/kg]\n",
                (0.00299 * m_fOverWidth * m_fOverHeight - 0.000832),
                (0.00513 + 17.6/fCarM) );
    cout << buf;
    sprintf( buf, "\n" );
    cout << buf;
    sprintf( buf, " Number of gear = %2d\n", m_nMaxGear );
    cout << buf;
    sprintf( buf, " gear   ratio  efficiency  torq margin    DW[kg]\n");
    cout << buf;

    for( p_ExceedForce = m_ExceedForce.begin();
                p_ExceedForce != m_ExceedForce.end();
                p_ExceedForce++ ){
      fDW = (m_fMFact + m_fEFact * p_ExceedForce->fGearti * p_ExceedForce->fGearti) * m_fCarIniW;
      sprintf( buf, " %3d:   %6.3f    %6.3f      %6.3f   %12.5f \n",
                p_ExceedForce->nGear,
                p_ExceedForce->fGearti,
                p_ExceedForce->fForcePer,
                p_ExceedForce->fFreePer,
                fDW );
      cout << buf;
    }
    sprintf( buf, " fin:   %6.3f    %6.3f\n", m_fLastReduceGear, m_fUd );
    cout << buf;
    sprintf( buf, "\n" );
    cout << buf;

}
/**/
/*********************************************************************
 * Function name          : WriteAllCalculateData
 * Function summary       : Processed data output processing
 * Explanation            : Processing result is output to file.
 *                        :
 * Argument (input)       : None
 * Argument (output)      : None
 * Argument (I/O)         : None
 * Return value           : OK : Normal   NG : Failure
 * Created by             :
 * Updated on (created on) :
 * Remarks                :
 *********************************************************************/
int   TCalculateProc::WriteAllCalculateData()
{
    string     szTmp,szFile;
    int        nRet;
    char       buf[1024];
    FILE       *m_pFile;
    double     fMaxTe;
    bool       tmpbTe_f;
    bool       tmpbN_norm_f;
    bool       tmpbT_norm_f;
```

```
double      tmpfVref;
double      tmpfVana;
double      tmpfNe;
double      tmpfTe;
double      tmpN_norm;
double      tmpT_norm;
char        tmp_strfTe[128];
char        tmp_strfNe[128];
char        tmp_strN_norm[128];
char        tmp_strT_norm[128];

GetCalculateDataFileName(szFile);

if( ( m_pFile = fopen( szFile.c_str(), "wt" ) ) == NULL ){
    sprintf( buf, "%s\n\nThe file is not found.",
            szFile.c_str() );
    cout << buf << endl;
    return( NG );
}

nRet = WriteHead(m_pFile);
if (nRet != OK){
    return( NG );
}

for( p_setCalculateData = setCalculateData.begin();
        p_setCalculateData != setCalculateData.end();
        p_setCalculateData++ ){
    if( p_setCalculateData->second.nWriteFlg != 1 ){
        continue;
    }
    fMaxTe = GetLineReviseMaxTorque(p_setCalculateData->second.fNegrevo);

    tmpfTe    = p_setCalculateData->second.fTe;
    tmpN_norm = (((p_setCalculateData->second.fNegrevo - m_fIdleNe)/( m_fFixedNe - m_fIdleNe )) * 100.0);
    if( fMaxTe <= 0 ){
        tmpT_norm = 0;
    }else{
        tmpT_norm = ((p_setCalculateData->second.fTe / fMaxTe) * 100.0);
    }

    tmpbTe_f = false;
    tmpbN_norm_f = false;
    tmpbT_norm_f = false;
    if( tmpfTe < 0 ){
        tmpbTe_f = true;
    }
    if( tmpN_norm < 0 ){
        tmpbN_norm_f = true;
    }
    if( tmpT_norm < 0 ){
        tmpbT_norm_f = true;
    }

    tmpfVref = p_setCalculateData->second.fVref_sp;
    tmpfVana = p_setCalculateData->second.fVana_sp;
    tmpfNe   = p_setCalculateData->second.fNegrevo;

    if( tmpbTe_f == false ){
        sprintf( tmp_strfTe, "%f", tmpfTe );
        tmp_strfTe[strlen(tmp_strfTe)-1] = 0x00;
        tmpfTe = atof( tmp_strfTe );
        sprintf( tmp_strfTe, "%.1f", tmpfTe );
    }else{
        sprintf( tmp_strfTe, "%s", "M" );
    }
    sprintf( tmp_strfNe, "%f", tmpfNe );
    tmp_strfNe[strlen(tmp_strfNe)-1] = 0x00;
    tmpfNe = atof( tmp_strfNe );
    sprintf( tmp_strfNe, "%.1f", tmpfNe );
    if( tmpbN_norm_f == false ){
        sprintf( tmp_strN_norm, "%.2f", tmpN_norm );
    }else{
        sprintf( tmp_strN_norm, "%s", "M" );
    }
    if( tmpbT_norm_f == false ){
        sprintf( tmp_strT_norm, "%.2f", tmpT_norm );
```

```
        }else{
            sprintf( tmp_strT_norm, "%s", "M" );
        }

        sprintf(buf,"%d\t%.2f\t%.2f\t%s\t%s\t%s\t%s\t%d",
                (int)(p_setCalculateData->second.fTimes /10),                    // Accumulated time
                tmpfVref,                                                        // Reference vehicle speed
                tmpfVana,                                                        // Analysis vehicle speed
                tmp_strfNe,                                                      // Engine speed
                tmp_strfTe,                                                      // Engine torque
                tmp_strN_norm,
                tmp_strT_norm,
                p_setCalculateData->second.nGear );                              // Gear position

        nRet = fprintf(m_pFile, "%s\n", buf);
        if (nRet == EOF){
            fclose(m_pFile);
            cout << MSG_WRITE_FILE_ERROR << endl;
            return( NG );
        }
    }

    fclose(m_pFile);
    return( OK );
}
/**/
/*********************************************************************
 * Function name          : WriteHead
 * Function summary       : Analysis data header output processing
 * Explanation            : Header is output to processing result file.
 *                        :
 * Argument (input)       : *fp : Analysis data output file name
 * Argument (output)      : None
 * Argument (I/O)         : None
 * Return value           : OK : Normal   NG : Failure
 * Created by             :
 * Updated on (created on) :
 * Remarks                :
 *********************************************************************/
int   TCalculateProc::WriteHead(FILE *fp)
{

    int nRet;
    string  szFieldTitle;

    szFieldTitle = DEF_PRINT_POS1;
    szFieldTitle = szFieldTitle + "\t" + DEF_PRINT_POS2;
    szFieldTitle = szFieldTitle + "\t" + DEF_PRINT_POS3;
    szFieldTitle = szFieldTitle + "\t" + DEF_PRINT_POS4;
    szFieldTitle = szFieldTitle + "\t" + DEF_PRINT_POS5;
    szFieldTitle = szFieldTitle + "\t" + DEF_PRINT_POS6;
    szFieldTitle = szFieldTitle + "\t" + DEF_PRINT_POS7;
    szFieldTitle = szFieldTitle + "\t" + DEF_PRINT_POS8;

    nRet = fprintf(fp, "%s\n", szFieldTitle.c_str());
    if (nRet == EOF){
        cout << MSG_WRITE_FILE_ERROR << endl;
        return( NG );
    }

    return( OK );
}


/**/
/*********************************************************************
 * Function name          : TCommFun
 * Function summary       : Constructor
 * Explanation            : Class constructor
 *                        :
 * Argument (input)       : None
 * Argument (output)      : None
 * Argument (I/O)         : None
 * Return value           : None
 * Created by             :
 * Updated on (created on) :
 * Remarks                :
```

```
 ********************************************************************/
TCommFun::TCommFun()
{

}

/**/
/*********************************************************************
 * Function name          : ~  TCommFun
 * Function summary       : Destructor
 * Explanation            : Class destructor
 *                        :
 * Argument (input)       : None
 * Argument (output)      : None
 * Argument (I/O)         : None
 * Return value           : None
 * Created by             :
 * Updated on (created on) :
 * Remarks                :
 *********************************************************************/
TCommFun::~  TCommFun()
{

}
/**/
/*********************************************************************
 * Function name          : AStrToDouble
 * Function summary       : Comparison of two floating point values
 * Explanation            : Character string numeric is converted to floating point numeric.
 *                        :
 * Argument (input)       : szData : Character string numeric
 * Argument (output)      : fData  : Floating point
 * Argument (I/O)         : None
 * Return value           : true : Normal    false : Failure
 * Created by             :
 * Updated on (created on) :
 * Remarks                :
 *********************************************************************/
bool   TCommFun::AStrToDouble(string szData, double &fData)
{

   try
   {
      Trim(szData);
      if (!szData.empty())
      {
         fData = atof(szData.c_str());
      }else
      {
         return false;
      }
      return true;
   }

   catch (...)
   {
      return false;
   }

}
/**/
/*********************************************************************
 * Function name          : Trim
 * Function summary       : Character string truncation
 * Explanation            : Character string numeric is converted to floating point numeric.
 *                        :
 * Argument (input)       : None
 * Argument (output)      : None
 * Argument (I/O)         : str : Character string truncated/to be truncated
 * Return value           : None
 * Created by             :
 * Updated on (created on) :
 * Remarks                :
 *********************************************************************/
void   TCommFun::Trim( string &str )
{
   string tmpStr;
```

```
    int first_wd;
    int last_wd;

    //-------------------------
    // Remove space.
    //-------------------------
    first_wd = (int)(str.find_first_not_of( ' ', 0 ));
    last_wd  = (int)(str.find_last_not_of( ' ', str.size() ));

    tmpStr = str;
    if(( first_wd >= 0 )&&( last_wd >= 0 )&&( first_wd < last_wd )&&
       ( last_wd < (int)(str.size()) )){
        tmpStr = str.substr( first_wd, last_wd - first_wd +1);
    }

    str = tmpStr;

    //--------------------------------
    // Remove TAB.
    //--------------------------------
    first_wd = (int)(str.find_first_not_of( '\t', 0 ));
    last_wd  = (int)(str.find_last_not_of( '\t', str.size() ));

    tmpStr = str;
    if(( first_wd >= 0 )&&( last_wd >= 0 )&&( first_wd < last_wd )&&
       ( last_wd < (int)(str.size()) )){
        tmpStr = str.substr( first_wd, last_wd - first_wd +1 );
    }

    str = tmpStr;

    //--------------------------------
    // Remove line feed.
    //--------------------------------
    first_wd = 0;
    last_wd  = (int)(str.find_last_not_of( '\n', str.size() ));

    tmpStr = str;
    if(( first_wd >= 0 )&&( last_wd >= 0 )&&( first_wd < last_wd )&&
       ( last_wd < (int)(str.size()) )){
        tmpStr = str.substr( first_wd, last_wd - first_wd +1 );
    }

    str = tmpStr;
}
/**/
/*********************************************************************
 * Function name          : FileExists
 * Function summary       : File confirmation processing
 * Explanation            : Existence of specified file is checked.
 *                        :
 * Argument (input)       : filename : File to be checked
 * Argument (output)      : None
 * Argument (I/O)         : None
 * Return value           : true : Existing    false : Non-existing
 * Created by             :
 * Updated on (created on) :
 * Remarks                :
 *********************************************************************/
bool   TCommFun::FileExists( string filename )
{
    FILE *fp;

    fp = fopen( filename.c_str(), "r");
    if(( fp == NULL )||( ferror(fp) )){
        cout << "File Not Found. [" << filename << "]" << endl;
        return(false);
    }
    fclose(fp);
    return(true);
}


/**/
/*********************************************************************
 * Function name          : main
 * Function summary       : Main processing
```

```
 * Explanation             : Main process of conversion processing
 *                         :
 * Argument (input)        : None
 * Argument (output)       : None
 * Argument (I/O)          : None
 * Return value            : None
 * Created by              :
 * Updated on (created on) :
 * Remarks                 :
 ************************************************************************/
int main(int argc, char* argv[])
{
    int  nRet;
    bool bRet;

    string tmpStr;
    string tmpFileName;

    // Initialization
    CommFun     = new TCommFun();
    CalculateProc = new TCalculateProc();

    if( argc >= 2 ){
        tmpFileName = string( argv[1] );
        bRet = CalculateProc->Init( tmpFileName );     // Reads environmental data from file.
    }else{
        bRet = CalculateProc->Init();                  // Reads environmental data from file.
    }
    if( bRet == false ){
        cout << "Stopped with error." << endl;
        exit(-1);
    }

    // Conversion processing initiation
    cout << "Convert start!" << endl;
    nRet = CalculateProc->CalculateProcess();   // Initiates conversion processing.
    if( nRet == NG ){
        cout << "Stopped with error." << endl;
        exit(-1);
    }

    cout << "Convert finished!" << endl;

    // Post-processing
    delete CommFun;
    delete CalculateProc;

    exit(0);
    return(0);
}
//---------------------------------------------------------------------------
#endif
```