General Thermodynamic Package (GTP)

Basic description and documentation

Bo Sundman, January 9, 2016

This is a preliminary description of the data structure and subroutines in the thermodynamic model package GTP. A recent paper describing this initiative to develop a free thermodynamic software can be found in[15Sun1].

This documentation is neither complete nor up to date with the most resent version. The whole structure (and depending subroutines) will be revised when all major parts have been implemented and tested.

The software is written in the new Fortran standard and the reason to use Fortran is that it is specially designed for numerical calculations and there are many numerical libraries available. It is also useful for massive parallel processing. Extensive use is made of the datatyping available in Fortran08.

Documentation updating software

One very difficult thing with software that is alive and changing is to update the documentation. So my feeling for documentation is that once a software is documented it is dead as it is almost impossible to update source code and documentation in parallel.

To handle this I have written a simple documentation update software that uses a basic documentation written as a LaTeX file together with the source code to extract verbatim sections. When running this software with the current documentation file and a new source code it compares the existing verbatim sections with those it can find in the new source code and flaggs all differences. This is a simple method to find what has changed in the source code.

The verbatim sections in the source code is enclosed by

```
!\begin{verbatim}
extract from source code
!\end{verbatim}
```

These can be inserted anywhere in the source code. Normally they enclose important sections like data structure definitions and declarations of subroutines and functions. The documentation software finds new or changed verbatim sections as well as verbatim sections that has disappeared and writes a new LaTeX documentation file which require editing to describe the changes and to add crossreferences. Using this software I hope to keep the software documented also during the development stage. This document is an example of this software.

The subroutines are documented in the order they appear in the source code. That order is not always logical and a major restructuring will be made for the next release (I hope at least). I would also like to add a table with the subroutines in alphabetical order. But one thing at a time.

Contents

1 The Gibbs energy function

In a thermodynamic system one has elements, species (combination of elements with fixed stoichiometric ratios and a possible charge) and phases. The thermodynamic properties of the system are described by models for the Gibbs energy which can be different for each phase. The Gibbs energy for a phase depend on the temperature, T, pressure, P and amounts of the constituents of the phase. The constituents are a subset of the species.

At equilibrium at constant T, P and composition the Gibbs energy of a system is at a minimum. With suitable transformations the Gibbs energy can be used to find the equilibrium of a system for any other set of conditions. The model function can also be used to extrapolate to non-equilibrium states. Some thought has been spent on modeling special physical properties like ferromagnetism and additional external variables like Curie Temperature as elastic constants can be included in the modeling.

The method of storing composition dependant properties used for the Gibbs energy parameters is suitable also for other properties like mobilities, resistivity, viscosities etc and there are provisions to include such data also in the database format.

The use of this model package to calculate equilibria in multicomponent systems and for calculation of property and phase diagrams is explained in a separate document.

1.1 New features in current and previous versions

The release of version 3 of OC is scheduled to the end of 2015. It includes several improvements in the already existing software but most important a new assessment procedure to fit model parameters to experimental and theoretical data. This has added a few new data structures and commands to the GTP package. Version 3 can also be used for parallel calculations of several equilibria using the OpenMP package. A significant increase in speed has been noted.

The previous version 2 of OC was released in February 2015 and the main new feature of this version was the step and map procedures which are documented separately. It also included derivatives of state variables, "dot derivatives", and some improvements of the minimization procedure and main change in the GTP module was implementing the ionic liquid model. This did not really change much in this documentation, the code for handling the variable ratio of sublattices was been integrated in the existing subroutines.

1.2 Features to be added in future versions

A software is never finished but each version has to be tested and debugged before new features can be added. A feature that is still missing in the GTP package is the

possibility to save the data structures and calculated results on a random access file that can be later read back and used for further calculations.

In the GTP package several new models for phases will be implemented in future versions including models for high pressure. Models related to describing heat capacities down to zero Kelvin, for the glass transition are also on the list. As this software is free it is possible for any interested scientist to contribute.

1.3 The Gibbs energy function

The model package provides the possibility to calculate the Gibbs energy, G, for many phases for any value of T, P and its constitution using the models implemented. In addition it can calculate the first and second derivatives of G with respect to T, P and the constitution. These values can be used directly or within a minimization package developed in parallel, to handle equilibrium calculations in multicomponent system with several phases for various external conditions. The basic formula for the Gibbs energy of a phase, independent of the model, is taken from the book by Lukas, Fries and Sundman[07Luk].

$$G_M = {}^{\mathrm{srf}}G_M - T\ {}^{\mathrm{cfg}}S_M + {}^{E}G_M + {}^{\mathrm{phys}}G_M \tag{1}$$

where ${}^{\mathrm{srf}}G_M$ is the surface of reference containing parameters for the endmembers, ${}^{\mathrm{cfg}}S_M$ is the configurational entropy, ${}^{E}G_M$ is the excess Gibbs energy which has interaction parameters within the phase and ${}^{\mathrm{phys}}G_M$ is the contribution to the Gibbs energy due to some physical properties, like ferromagnetism, that are modeled separately.

The Gibbs energy is modeled for a formula unit of the phase, the formula unit defined by the crystallography, and the amount of the components for a formula unit of a phase can vary because the constituents can be for example molecules in a gas phase or vacancies occupying vacant lattice sites. The composition dependence of the modeled Gibbs energy depend on the constituent fractions, denoted y_i, possibly with a sublattice index and the mole fractions of a component A in the phase α can be calculated using the formula:

$$x_{\mathrm{A}}^{\alpha} = \frac{\sum_s a_s^{\alpha} \sum_j b_{\mathrm{A}j}\ y_{js}^{\alpha}}{\sum_s a_s^{\alpha} \sum_{\mathrm{B}} \sum_j b_{\mathrm{B}j}\ y_{js}^{\alpha}} \tag{2}$$

where a_s^{α} is the number of sites on sublattice s, $b_{\mathrm{A}j}$ is the stoichiometric ratio of element A in species j and y_{js}^{α} is the constituent fraction of species j on sublattice s. The summation over B is for all components. The sum of site fractions in a sublattice s is unity:

$$\sum_i y_{is} = 1 \tag{3}$$

The terms and factors used in this equation are explained below and for further details please read the book by Lukas et al.

2 Basis

The basic terms and concepts of the GTP model package are explained here. Depending on your background some definitions my not agree with what you are used.

2.1 Elements

The elements have a sequential index arranged alphabetically. They have a one or two-letter symbol and a mass. The symbol must be letters A-Z all in upper case. In addition a "name" of the element is stored, like "iron" for Fe, the name of a reference phase, the value of H298-H0 for the reference phase and S298 for the reference phase. The element symbol must be unique.

OC is case insensitive, it means that an element written as Fe is the same as fe or FE or fE. The element Cobalt can be written co and to specify the species carbon monoxide we must write c1o or c1o1 if there is also c1o2 in the system.

The electron and vacancy are entered as element -1 and 0 respectively but they are not really treated as elements except that they can be included in species. The electron has the symbol /- but one can also use /+ to denote a positive charge (or /–1). Vacancies are denoted "Va" and have no fixed amount and will always have zero chemical potential at equilibrium. They will thus not affect the Gibbs phase rule.

An element can be be suspended, that means it can be hidden from application programs. If an element is suspended then all species that include this element are also suspended (implicitly) and also any phases which cannot exist unless the species is present as constituent, i.e. the species in the only constituent in a sublattice.

2.2 Species

The species are stoichiometric combinations of elements. They have a symbol and a stoichiometric formula. The elements, including the vacancy, are the simplest species. The stoichiometry can be non-integer like a species $ALO_{1.5}$. The symbol must start with a letter A-Z and can contain letters (case insensitive), digits, the period ".", the underscore character "_", the slash "/", the minus sign "-" and the plus sign "+". No other special characters are allowed, for example parenthesis are not allowed.

The symbol of the species is often the same as the stoichiometric formula but it is not necessary. For example the molecule $C_2H_2Cl_2$ exists in 2 configurations either with

CL on the same side or opposite. These can be identified by giving these the names C2H2CL2_CIS and C2H2CL2_TRANS.

In addition the mass is stored but this is redundant as it is calculated from the element masses. The species can have a charge like Al/+3 or O/-2. The charge can be non-integer.

Note that a species does not represent a phase, just a stoichiometric formula. As constituents of a phase the species are associated with that phase. The species H2O can be a constituent of ice, liquid water and gas. Thus species have no thermodynamic data. Their only function is to be constituents of phases where they can be assigned data. A species must not be confused with a phase just because many phases have a fixed stoichiometric formula. In this package the same species can be a constituent of many phases, also phases with variable composition.

When entering the stoichiometric formula for a species one may need to use stoichiometric numbers to separate one letter elements as OC is case insensitive. For example to distinguish CO as cobalt from C1O1 as carbon monoxide the digit 1 can be used after the element letter C. But otherwise the stoichiometric number 1 is not needed if the element have a two letter symbol, for example CAO can be used for a species with one atom of CA and one atom of O. One cannot use parenthesis in species symbols or when entering the stoichiometry, the stoichiometry of $Ca(OH)_2$ must be entered as CAO2H2 (or O1CAO1H2 or any other way giving the same stoichiometry).

The fact that a species has a symbol which is independent of its stoichiometry can be used to simplify complicated chemical stoichiometries.

It is important to keep in mind that a species is identified with its symbol, not its stoichiometry. Thus a species with the symbol c1o1 cannot be found by giving the symbol o1c1.

2.3 Phases

All thermodynamic data that are used in calculations are stored as part of a phase. The Gibbs energy of the phase is modeled as a function of temperature (T), pressure (P) and the constitution of the phase (Y) as given by eq. 1. Many different thermodynamic models can be used and each phase can have a unique model and also several different "additions" to this model, like magnetic and pressure models. But all phases are accessed from outside the model package in the same way. Thus the minimizer which is used to find the equilibrium state of a system does not depend on any particular model feature.

From outside the model package one can obtain information of the constitution of the phase, i.e. which constituents and their fractions (Y). The constituents are species (with fixed stoichiometry). As the compound energy formalism (CEF) is a very general and flexible model framework this has been the first implemented. In version 2 of OC

the ionic liquid model is also implemented. There is no special documentation of that model in this text as it is described in the book by the book by Lukas, Fries and Sundman [07Luk].

In CEF one can define sublattices with specific species (elements or molecules) as constituents and the constituent fractions on each sublattice must be unity. The constituents are assumed to mix randomly on each sublattice. For a general description of CEF please consult Lukas et al.[07Luk].

One may implement models that do not use sublattices and have non-random entropy in GTP, for example CVM or quasichemical models. Inside the CEF framework this can be done by treating the phase as having a single lattice and the probabilities of the different clusters are fractions and the clusters are species. Inside the model package the appropriate non-random configurational entropy is then evaluated. The reason to base the model package on the constitution and not the composition of the component is to avoid a two-level minimization when calculating the equilibrium. But there is nothing, except speed, preventing from using the mole fractions as external variables for some models and implement an internal minimization to determine the configuration which will give the minimum Gibbs energy for the given external conditions.

There are some models (not implemented in OC) which allow a variable stoichiometry of the constituents of the phase. Such models must be implemented as a two-level minimization where the model from outside the model package depend only the mole fractions of the elements or some other set of components. It may also be necessary to have some special way of entering parameters for such a model.

2.3.1 Phase specification

A phase has a name and some model information. If it is a CEF model it has number of sublattices, the sites in each sublattice (a fixed real value) and a list of constituents on each sublattice. One may have a single constituent in a sublattice. An empty sublattice can have the vacancy as the only constituent.

The term "endmember" refers to a combination of one constituent in each sublattice. This defines a compound and this is the origin of the name "compound energy formalism". When writing an endmember the constituents are separated by a colon.

The thermodynamic model parameters are entered for the endmembers and possible interaction parameters.

The ionic liquid model has variable number of sites for cations and anions to maintain electro neutrality. This must be taken care of when calculating the mass balance during with the minimization.

When entering an endmember parameter the software writes the amount and reference states of the elements this is referred to. Care must be taken that the parameter value correspond to the correct amount of atoms as this can vary between endmem-

bers of the same phase. For example if the FCC phase is defined with 2 sublattices with one site in each, the endmember G(FCC,TI:VA) is for one mole of atoms whereas G(FCC,TI:C) is for two moles of atoms.

2.3.2 Interaction parameters

For interaction parameters one must add one, two or three additional constituents in any sublattice separated by a comma from the other constituents in the same sublattice. The fractions of the constituents specified in the parameter should be multiplied with the parameter value.

An interaction parameter may also have a degree specifying that it has a more complex composition dependence. The degree is a digit 0 to 9 given after a semicolon. The default degree is zero. The meaning of the degree depend on the model, for a binary interaction in CEF it is by default the power in a Redlich-Kister (RK) series:

$$L_{ij} = \sum_{\nu=0} (y_i - y_j)^\nu \ {}^\nu L_{ij} \tag{4}$$

where y_i and y_j are the constituent fractions on the same sublattice and ν is the degree of the parameter. For $\nu = 0$ the parameter is composition independent.

Each such RK coefficient is a function of temperature and pressure. There can also be different extrapolation models like Muggianu, Kohler and Toop but they are not implemented yet.

For a ternary parameter there is also possibilities to have composition dependence:

$$\begin{aligned} L_{ijk} &= v_i \ {}^0 L_{ijk} + v_j \ {}^1 L_{ijk} + v_k \ {}^2 L_{ijk} && (5)\\ v_i &= y_i + (1 - y_i - y_j - y_k)/3 && (6)\\ v_j &= y_j + (1 - y_i - y_j - y_k)/3 && (7)\\ v_k &= y_k + (1 - y_i - y_j - y_k)/3 && (8) \end{aligned}$$

where y_i, y_j and y_k are the constituent fractions on the same sublattice. The v_i are the same as y_i in the ternary system and the additional terms have been added to make this contribution symmetrical in higher order systems.

2.3.3 Model parameters identifiers

In addition to calculate the Gibbs energy for a phase the user can also enter parameters for other properties. These parameters can depend on T, P and the constitution of the phase. In this package all parameters are identified by

- a property symbol, listed below

- the phase name,
- the constituent array (specifying the constituents in each sublattice, the fraction of which are multiplied with the parameter), and
- a degree with a value 0 to 9,

for example G(FCC,FE), G(SIGMA,FE:CR:CR), G(LAVES,FE,MO:MO;2), TC(BCC,FE) or MQ&FE(FCC,FE), see also 2.3.5.

The property symbol G is used for the Gibbs energy. Other properties symbols like TC is used for the Curie temperature and MQ&XY for the mobility of the element XY the the phase.

This means it is possible to include model properties that does not give any contribution to the Gibbs energy, like the mobility, and thus the model package can handle the composition dependence of many properties like resistivity, viscosity etc. A tentative list of properties that have been defined is given in Table 1.

Table 1: Parameter identifiers

Symbol	T	P	Specification	Meaning
G	T	P		Gibbs Energy
TC	-	P		Mixed Curie/Neel T
BMAG	-	-		Aver Bohr magn number
CTA	-	P		Curie temperature
NTA	-	P		Neel temperature
IBM	-	P	&¡constituent#sublattice¿	Individual Bohr magn number
THET	-	P		Debye or Einst temp
MQ	T	P	&¡constituent#sublattice¿	LN mobility of const
RHO	T	P		Elect resistivity
MAGS	T	P		Magn susceptibiltyi
GTT	-	P		Glass trans temp
VISC	T	P		Viscosity
LPX	T	P		Lattice parameter
EC11	T	P		Elast const C11
EC12	T	P		Elast const C12
EC44	T	P		Elast const C44

In case the parameters should affect the Gibbs energy, like the Curie temperature, an "addition" subroutine must be written using the value of this parameter. See section 5.6.9.

The TC and BMAG parameters are included for compatibility with the current magnetic model, most of the others are just tentative.

2.3.4 Physical models

Traditionally CALPHAD models describes the variation of the Gibbs energy as a function of T, P and composition without bothering about the origin of the contributions. In some cases this is no so good and for the contribution due to ferromagnetic transitions one has introduced a more physical contribution that depend on two new variables, the Curie temperature and the Bohr magneton number, both of which depend on the constitution. Such *additions* to the traditional Gibbs energy expression may become more frequent when also phonon contributions are modeled down to 0 K as a function of a constitution dependent Debye temperature. Hopefully this can lead to better extrapolations to higher temperatures as well as multicomponent systems.

2.3.5 Other properties than the Gibbs energy

The Gibbs energy parameters has property index 1 (one). There are a number of predefined properties in the code, that is the Curie temperature (index 2), average Bohr magneton number (3) etc. These are defined in the subroutine init_gtp and a tentative list of predefined identifiers is given in Table 1. More types can be added but one must choose a unique symbol for each and each is assigned a sequential index. The symbol must not be mistaken for a state variable symbol either. In the property records, see 5.6.6, for endmembers and interactions this propery index is used to identify the property.

The mobility of an element in a phase depends on the constitution and the temperature but the mobility does not contribute to the Gibbs energy. Anyway it is possible to define and enter mobilities in GTP to handle their composition dependence. The values of these, and any other separate property like the Curie temperature, can be obtained anytime by a special subroutine call.

One must consider that the mobility for each constituent depend individually on the constitution and a symbol MQ&X(BCC) is used to define the mobility of X in the BCC phase as a function of its constitution. Thus MQ&FE(BCC,FE) is the self diffusion of FE in BCC and MQ&FE(BCC,CR) is the mobility of FE in pure BCC Cr. If there are data one can have interaction parameters like MQ&FE(BCC,CR,FE). Each mobility parameter can depend on T and P.

Other properties that may be described by this model package are lattice parameters, elastic constants, resistivity, etc. see Table 1.

2.4 Components

The term "component" has no meaning for the modeling, only for controlling the system externally. The species that can dissolve in a phase are called constituents. The term component is reserved for an irreducible set of species that describe the

composition space of the system, sometimes called "system components". In most cases these components are the elements. For many simple models the constituents of the phases are also identical to the elements.

The components are important as they determine the number of conditions that can be set in order to calculate the equilibrium of the system because one can only set as many conditions as one has components and for T and P. Many conditions, like the amounts or chemical potentials, can only be specified for the components, not for an individual species. This does not mean that the calculation of equilibria is limited to conditions on the amounts of the components, only that the number of components determine the number of conditions that must be set in order to calculate the equilibrium.

In GTP the elements are by default the components but the user may change this to any orthogonal set of species. When working with oxides with no degree of freedom for the oxygen content, like in a quasibinary CaO-SiO_2 system, it may be convenient to select the oxides CaO and SiO_2 as components and then leave the content of the third element undetermined by setting an arbitrary (positive) value of its activity. But in such cases it is necessary to know that the models used for the phases does not allow the content of this element to vary independently. If a gas phase is included in the system it may be more convenient to set the gas as fixed with zero amount, although that may increase the calculation time.

2.5 Fraction sets

In some software like Thermo-Calc (TC) one may have a phase described by two separate Gibbs energy functions, this is called "partitioned" or "splitted". It was originally used for phases with order/disorder transformations like A1/$L1_2$ or A2/B2 where the phase can be completely disordered. All parameters to describe the disordered state where collected in a phase that had no sublattices for ordering (it could have a sublattice for interstitials). This made it also easier to include ordering in multicomponent databases where many subsystems have A1 (fcc) or A2 (bcc) phases without ordering.

The Gibbs energy of a partitioned phase used to be written in Thermo-Calc:

$$G_M = G_M^{\rm dis}(x) + \left(G_M^{\rm ord}(y) - G_M^{\rm ord}(y=x)\right) \quad (9)$$

$$\Delta G_M^{\rm ord} = G_M^{\rm 4SL}(y) - G_M^{\rm 4SL}(y=x) \quad (10)$$

The parameters for the ordered phase, as disordered, was included in the disordered phase.

There is no reason to do this with the OC software as the disordered parameters are included in the same phase as the ordered. The Gibbs energy for a partitioned ohase is simply:

$$G_M = G_M^{\rm dis}(x) + G_M^{\rm 4SL}(y) \quad (11)$$

where the parameters in the disordered part only has parameters that describe the disordered contribution, independent of the parameters for the ordering.

The implementation in Thermo-Calc version S and maybe later require that the parameters for the ordered part, as disordered, must be included in $G^{\rm dis}(x)$. That is not the case for OC.

A variant of this model have been applied also to intermetallic phases that never disorder completely like σ, μ, Laves phase and similar. The equation is then slightly different as there is no need to describe the completely disordered state

$$G_M = \left(G_M^{\rm dis}(x) + T\ {}^{\rm cfg}S_M^{\rm dis}(x)\right) + G_M^{\rm ord}(y) \tag{12}$$

By adding $T\ {}^{\rm cfg}S_M^{\rm dis}(x)$ the configurational entropy is only calculated for the ordered part.

This second variant of the disordered model is very interesting because many intermetallic phases like the σ phase have a very large number of end members in multicomponent systems and they must all be given a reasonable value, although in databases today most of them are just equal to the sum of the reference energies of the endmembers, i.e. they can be set to zero. With the increased use of first principles calculations for ordered endmembers, only those which will decrease the Gibbs energy need to be included.

In the OC software we have taken this into account from the beginning and due to the parameters in the disordered part it is not necessary to enter all endmembers of the ordered part, only those which is known to be stable or close to be stable. *Ab initio* data can be used for this when experimental data are missing. The disordered part is entered as an extra "fraction set" of the ordered phase and one simply specifies how many sublattice that should be added together to for the disordered part. For an fcc phase with 5 sublattices, 4 for ordering and one interstitial, one simply gives 4, assuming the 5th sublattice is the interstitial one. For a σ phase modeled with 3 sublattices that all disorder together, one gives 3.

For a ternary σ phase with 5 sublattices one have more than 250 endmembers which all must be calculated. With the new software this may be reduced to less than 10 which will result in a significant increase in calculation speed compared to the case when all endmembers have a value. The maintanance of the database will also be simplified as fewer endmembers are stored.

The new software will create a fraction set record that gives the way to add fractions (the coefficients b_{ij} are stored there as they are also needed to calculate the contribution to the partial derivatives). A link to this fraction set record is stored in the phase_varres record linked from the phase record. When setting the constitution of the phase, i.e. the site fractions, the disordered fractions will be automatically calculated and saved in another phase_varres recored linked from the ordered fraction set record. In this way it will thus be straightforward to calculate the ordered and disordered part with

the same subroutine, one just have to organize the calculations of each part separately and then add all values, including first and second derivatives, together in the end.

2.6 Composition sets and miscibility gaps

Composition sets is the way OC handles the case when a phase with the same structure can appear simultaneously with two or more compositions. One case this is necessary is when a phase split up into two or more miscibility gaps like the bcc phase in the Cr-Mo system. Less obvious cases is when a phase has an order/disorder transition like the fcc phase in Au-Cu forms $L1_2$ and $L1_0$ structures. But also in that case the ordered and disordered phases are described with the same thermodynamic data. Finally this is technique is also used to describe the cubic and hexagonal carbides where the metallic atoms are the same as fcc and hcp lattices but interstitial atoms lice C and N can occupy most of the available interstitial sites.

Outside the OC package, for example in the TQ package, one can obtain information about the number of composition sets for a phase using the function "nocs(iph)" for each phase "iph". Inside the OC package the phase number and composition set number are usually given separately. A phase must have at least one composition set and the maximum number is 9. The composition sets larger than 1 is normally suffixed after the phase name separated by a # like BCC#2. If omitted composition set 1 is assumed.

If there are several composition sets of a phase there is a separate phase_varres record for each composition set. Normally these have identical set of constituents but it is possible to suspend constituents in each composition set separately. At the time when a fraction set is added to a phase this must not have any suspended constituents.

Fraction set records have no array or free list, they exist only inside the phase_varres records. (This created some extra headache when implementing them as a simple assignement of a record to another did not create any new record in Fortran08).

2.7 State variables

State variables are, as the name indicate, only dependent on the state of the system, not how this state was reached. In principle they have a well-defined value only at equilibrium but in the modeling they can be extended also outside this range.

The state variables recognized by GTP and that can be used are defined in Table 2. Many of them can have several indices and/or normalizing quantities. In the box below there are some examples of using these in conditions and otherwise.

N=1	means the system has one mole of atoms
N(H)=1	means the system has 1 mole of component H
N(GAS,H)=1	means the gas phase has 1 mole of component H
NP(LIQUID)=1	means the liquid phase has 1 mole of components
X(H)=0.3	means the mole fraction of component H in the system is 0.3
X(GAS,H)=0.3	means the gas phase has a mole fraction of 0.3 of H
B=1000	means the system has a mass of 1000 grams
W%(CR)=18	means the system has 18 mass percent of component CR
W(LIQUID,C)=0.01	means the liquid has 0.01 mass fraction of carbon
H	is the total enthalpy of the system
H(LIQUID)	is the enthapy of the current amount of liquid (can be zero)
HM(LIQUID)	is the enthalpy of liquid per mole components
HV(LIQUID)	is the enthalpy of liquid per m^3
HW(LIQUID)	is the enthalpy of liquid per gram
HF(LIQUID)	is the enthalpy of one formula unit of liquid

3 Fortran data structures

A number of arrays must be dimensioned, this is done by defining some constants (called PARAMETER in Fortran) and then using these to dimension arrays with fixed size. It may be possible to set these limits in a more flexible way by allocating the arrays in a special subroutine for initiation.

As I am learning Fortran08 by this programming project I discover new possibilities now and again. It seems possible to allow the initialization subroutine to do this dimensioning which means one may tailor the dimensioning of the arrays depending on the kind of calculation one will make.

3.1 User defined data types

The presentation of the data structures defined here follows the way they are declared in the Fortran source code. That may sometimes not be entirely logical but it makes it simpler to update the documentation.

In Fortran08 the programmer can define specific data types and several such are used to store the model information. A data type can contain many different kinds of variables like characters and numerical information. One can also have links between instances (records) of the same and different kinds of datatypes.

This kind of method to organize data is available in all modern computing languages and has been severely missed in Fortran. In Thermo-Calc it was handled by storing data in the self-designed workspace.

Some data types are straightforward to implement like the element. Thus one defines

Table 2: A very preliminary table with the state variables and their internal representation. Some model parameter properties are also included. The "z" used in some symbols like Sz means the optional normalizing symbol M, W, V or F.

Symbol	Id	Index 1	Index 2	Normalizing suffix	Meaning
Intensive properties					
T	1	-	-	-	Temperature
P	2	-	-	-	Pressure
MU	3	component	-/phase	-	Chemical potential
AC	4	component	-/phase	-	Activity
LNAC	5	component	-/phase	-	LN(activity)
Extensive properties					
U	10	-/phase#set	-	-	Internal energy for system
UM	11	-/phase#set	-	M	Internal energy per mole
UW	12	-/phase#set	-	W	Internal energy per mass
UV	13	-/phase#set	-	V	Internal energy per m^3
UF	14	phase#set	-	F	Internal energy per formula unit
Sz	2z	-/phase#set	-	-	entropy
Vz	3z	-/phase#set	-	-	volume
Hz	4z	-/phase#set	-	-	enthalpy
Az	5z	-/phase#set	-	-	Helmholtz energy
Gz	6z	-/phase#set	-	-	Gibbs energy
NPz	7z	phase#set	-	-	Moles of phase
BPz	8z	phase#set	-	-	Mass of phase
Qz	9z	phase#set	-	-	Stability of phase
DGz	10z	phase#set	-	-	Driving force of phase
Nz	11z	-/phase#set/comp	-/comp	-	Moles of component
X	111	phase#set/comp	-/comp	0	Mole fraction
X%	111	phase#set/comp	-/comp	100	Mole per cent
Bz	12z	-/phase#set/comp	-/comp	-	Mass of component
W	122	phase#set/comp	-/comp	0	Mass fraction
W%	122	phase#set/comp	-/comp	100	Mass per cent
Y	130	phase#set	const#subl	-	Constituent fraction
Some model parameter identifiers					
TC	-	phase#set	-	-	Curie temperature
BMAG	-	phase#set	-	-	Aver. Bohr magneton number
MQ&X	-	phase#set	constituent	-	Mobility of X
THET	-	phase#set	-	-	Debye temperature

a data type which has all the necessary information for an element and then allocates an array to hold as many elements that is necessary.

Other datatypes are more involved, like the record to hold information about an endmember parameter of a phase. This record must refer to the constituents that makes up the endmember and also a link to a list of properties that can be calculated for this end member. Finally it must be possible to link the endmember record in a list for a specific phase and it must be possible to have a link to interaction records based on this endmember. Before one can figure out how to store endmember data it is thus necessary to have a data structure containing many different kinds of data. One must also consider which data that may be different in different parallel processes dealing with the phase and which data that are "static". Many of these records may also be of different size for different phases and for example an endmember record for a phase with 3 sublattices need 3 times as many places for constituent indices than a phase with a single lattice. This can be handled with the dynamic allocation feature in Fortran08.

The TYPE definition feature in Fortran08 may be a little less flexible than in other programming languages but it also avoids some of the strange features of these like the "this" operator needed at strange places in some cases. There is no "new" operator either, records are usually declared as single entities or as arrays with fixed (although allocatable) size. But one can use pointers to allocate new records which can be found only by these pointers.

Some of the structures are used to define arrays, like the element, species and phases. Some are created dynamically and can only be accessed by pointers, like endmembers and property records. Some are used locally inside subroutines and declared globally only if used in several.

3.2 Bits of information

Many records contain information that is of YES/NO or ON/OFF type and they are often stored as bits in a status word. Some of these are set automatically and there are subroutines and commands to change others. They are summarized below for all records.

The use of each bit is explained in the text from the source code.

```
!-Bits in global status word (GS) in globaldata record
! level of user: beginner, occational, advanced; NOGLOB: no global gridmin calc
! NOMERGE: no merge of gridmin result, NODATA: not any data,
! NOPHASE: no phase in system, NOACS: no automatic creation of composition set
! NOREMCS: do not remove any redundant unstable composition sets
! NOSAVE: data changed after last save command
! VERBOSE: maximum of listing
```

```
! SETVERB: explicit setting of verbose
! SILENT: as little output as possible
! NOAFTEREQ: no manipulations of results after equilirum calculation
! XGRID: extra dense grid for all phases
! NOPAR: do not run in parallel
! >>>> some of these should be moved to the gtp_equilibrium_data record
  integer, parameter :: &
       GSBEG=0,     GSOCC=1,     GSADV=2,      GSNOGLOB=3, &
       GSNOMERGE=4, GSNODATA=5,  GSNOPHASE=6,  GSNOACS=7, &
       GSNOREMCS=8, GSNOSAVE=9,  GSVERBOSE=10, GSSETVERB=11,&
       GSSILENT=12, GSNOAFTEREQ=13, GSXGRID=14,GSNOPAR=15
!-----------------------------------------------------------------
!-Bits in element record
  integer, parameter :: &
       ELSUS=0
!-----------------------------------------------------------------
!-Bits in species record
! Suspended, implicitly suspended, species is element, species is vacancy
! species have charge, species is (system) component
  integer, parameter :: &
       SPSUS=0, SPIMSUS=1, SPEL=2, SPVA=3, &
       SPION=4, SPSYS=5
```

3.2.1 Phase record bits

The phase record has many bits specifying various things. Additionally the composition set record has other bits, most important if the composition set is ENTERED, meaning it can be stable, FIX, meaning it is a condition that it is stable, SUSPENDED, meaning it must not be stable, and DORMANT meaning it must not be stable but the driving force is calculated.

A user can change these bits but he/she cannot set the CSSTABLE bit which is only set if the phase is stable after a full equilibrium calculation.

```
!-Bits in phase record
! hidden, implictly hidden, ideal, no concentration variation (NOCV),
! Phase has parameters entered (PHHASP),
! F option (FORD), B option (BORD), Sigma ordering (SORD),
! multiple/disordered fraction sets (MFS), gas, liquid, ionic liquid,
! aqueous, dilute config. entropy (DILCE), quasichemical (QCE), CVM,
! FACT,  not create comp. sets (NOCS), Helmholz energy model (HELM),
! Model without 2nd derivatives (PHNODGDY2), Elastic model A,
! explicit charge balance needed (XCB), extra dense grid (XGRID)
! Subtract ordered part (PHSUBO)
  integer, parameter :: &
```

```
       PHHID=0,     PHIMHID=1,  PHID=2,    PHNOCV=3, &     ! 1 2 4 8
       PHHASP=4,    PHFORD=5,   PHBORD=6,  PHSORD=7, &
       PHMFS=8,     PHGAS=9,    PHLIQ=10,  PHIONLIQ=11, &
       PHAQ1=12,    PHDILCE=13, PHQCE=14,  PHCVMCE=15,&
       PHFACTCE=16, PHNOCS=17,  PHHELM=18, PHNODGDY2=19,&
       PHELMA=20,   PHEXCB=21,  PHXGRID=22,PHSUBO=23
!
!-------------------------------------------------------------
!-Bits in constituent fraction (phase_varres) record STATUS2
! CSDFS is set if record is for disordred fraction set, then one must use
!     sublattices from fraction_set record
! CSDLNK: a disordred fraction set in this phase_varres record
! CSDUM2 and CSDUM3 not used
! CSCONSUS set if one or more constituents suspended (status array constat
!     specify constituent status)
! CSORDER: set if fractions are ordered (only used for BCC/FCC ordering
!     with a disordered fraction set).
! CSABLE: set if phase is stable after an equilibrium calculation ?? needed
! CSAUTO set if composition set created during calculations
! CSDEFCON set if there is a default constitution
! CSTEMPAR set if created by grid minimizer and can be suspended afterwards
!       when running parallel
! CSDEL set if record is not used but has been and then deleted (by gridmin)
   integer, parameter :: &
        CSDFS=0,    CSDLNK=1,  CSDUM2=2,    CSDUM3=3, &
        CSCONSUS=4, CSORDER=5, CSABLE=6,    CSAUTO=7, &
        CSDEFCON=8, CSTEMPAR=9,CSDEL=10
```

3.2.2 Some more bits

In particular the gtp_equilibrium_data record there are some bits that specifies how the equilibrium calculation can be done, for example if the software can use global grid minimization to find start values.

For the parameter property record there are also bits specifying of the property may depend on T and P or if it has a constituent or component specifier (like the mobility).

```
!-Bits in constat array for each constituent
! For each constituent: is suspended, is implicitly suspended, is vacancy
   integer, parameter :: &
        CONSUS=0,CONIMSUS=1,CONVA=2
!-------------------------------------------------------------
!-Bits in state variable functions (svflista)
! SVFVAL symbol evaluated only explicitly (mode=1 in call)
! SVFEXT symbol external ???
```

```
! SVCONST symbol is a constant (can be changed with AMEND)
   integer, parameter :: &
        SVFVAL=0, SVFEXT=1, SVCONST=2
!-------------------------------------------------------------
!-Bits in gtp_equilibrium_data record
! EQNOTHREAD set if equilibrium must be calculated before threading
! (in assessment) for example if a symbol must be evaluated in this
! equilibrium before used in another like H(T)-H298
! EQNOGLOB set if no global minimization
! EQNOEQCAL set if no successful equilibrium calculation made
! EQINCON set if current conditions inconsistent with last calculation
! EQFAIL set if last calculation failed
! EQNOACS set if no automatic composition sets ?? not used !! see GSNOACS
! EQGRIDTEST set if grid minimizer should be used after equilibrium
   integer, parameter :: &
        EQNOTHREAD=0, EQNOGLOB=1, EQNOEQCAL=2, EQINCON=3, &
        EQFAIL=4,     EQNOACS=5,  EQGRIDTEST=6
!-------------------------------------------------------------
!-Bits in parameter property type record (gtp_propid)
! constant (no T or P dependence), only P, property has an element suffix
! (like mobility), property has a constituent suffix
   integer, parameter :: &
        IDNOTP=0, IDONLYP=1, IDELSUFFIX=2, IDCONSUFFIX=3
!-------------------------------------------------------------
!- Bits in condition status word (some set in onther ways??)
! singlevar means T=, x(el)= etc, singlevalue means value is a number
! phase means the condition is a fix phase
  integer, parameter :: &
       ACTIVE=0,SINGLEVAR=1,SINGLEVALUE=2,PHASE=3
!-------------------------------------------------------------
!- Bits in assessment head record status
! ahcoef means coefficients enetered
  integer, parameter :: &
       AHCOEF=0
!
! >>> Bits for symbols and TP functions missing ???
```

3.2.3 Phase status revision

A phase status can vary, the default is ENTERED which means that the phase can be stable or not depending on the conditions set by the user. The status values are defined as symbols according to the list below:

```
! some constants, phase status
  integer, parameter :: PHHIDDEN=-4
```

```
  integer, parameter :: PHSUS=-3
  integer, parameter :: PHDORM=-2
  integer, parameter :: PHENTUNST=-1
  integer, parameter :: PHENTERED=0
  integer, parameter :: PHENTSTAB=1
  integer, parameter :: PHFIXED=2
  character (len=12), dimension(-4:2), parameter :: phstate=&
       (/'HIDDEN      ','SUSPENDED   ','DORMANT     ','ENTERED UNST',&
         'ENTERED     ','ENTERED STBL','FIXED       '/)
```

The phase status are also available as character strings in the array phstate, defined above, for use in listings.

The HIDDEN status is questionable and not implemented. A phase with the HIDDEN status should not be included in the normal list of phases but stored in a separate list. If the status of a phase is changed to or from being HIDDEN the system should be re-initiated. The possibility to implement HIDDEN is kept as the current database interface does not allow selection of phases when reading from a database.

3.3 Dimensioning

Most arrays are dimensioned in the GTP package using constants. If one need to change them it is usually only in one place. Most of these constants are used in the init_gtp subroutine but also when temporary arrays are needed inside some other subroutines.

```
! Parameters defining the size of arrays etc.
! max elements, species, phases, sublattices, constituents (ideal phase)
  integer, parameter :: maxel=100,maxsp=1000,maxph=800,maxsubl=10,maxconst=1000
! maximum number of constituents in non-ideal phase
  integer, parameter :: maxcons2=100
! maximum number of elements in a species
  integer, parameter :: maxspel=10
! maximum number of references
  integer, private, parameter :: maxrefs=1000
! maximum number of equilibria
  integer, private, parameter :: maxeq=900
! some dp values, default precision of Y and default minimum value of Y
! zero and one set in tpfun
  double precision, private, parameter :: YPRECD=1.0D-6,YMIND=1.0D-30
! dimension for push/pop in calcg, max composition dependent interaction
  integer, private, parameter :: maxpp=1000,maxinter=3
! max number of TP symbols
  integer, private, parameter :: maxtpf=20*maxph
! max number of properties (G, TC, BMAG MQ%(...) etc)
```

```
  integer, private, parameter :: maxprop=50
! max number of state variable functions
  integer, private, parameter :: maxsvfun=500
! version number
! changes in last 2 digits means no change in SAVE/READ format
  character*8, parameter :: gtpversion='GTP-3.00'
  character*8, parameter ::   savefile='OCF-3.00'
```

3.3.1 User defined additions

The additions provided or added by programmers must be uniquely identified. This is an attempt to do that with those already available.

```
! The number of additions to the Gibbs energy of a phase is increasing
! This is a way to try to organize them.  Each addtion has a unique
! number identifying it when created, listed or calculated.  These
! numbers are defined here
  integer, public, parameter :: indenmagnetic=1
  integer, public, parameter :: debyecp=2
  integer, public, parameter :: weimagnetic=3
  integer, public, parameter :: einsteincp=4
  integer, public, parameter :: elasticmodela=5
  integer, public, parameter :: glastransmodela=6
! Note that additions often use extra parameters like Curie or Debye
! temperatures defined by parameter identifiers stored in gtp_propid
```

3.4 The global error code

The error code is the only member of the gtp_parerr record. In parallel execution each thread must have its own value and it is thus declared threadprivate.

```
  TYPE gtp_parerr
! This record contains the global error code.  In parallel processing each
! parallel processes has its own error code copied to this if nonzero
! it should be replaced by gtperr for separate errors in treads
     INTEGER :: bmperr
  END TYPE gtp_parerr
  TYPE(gtp_parerr) :: gx
! needed to have error code as private in threads
!$OMP  threadprivate(gx)
```

4 Data structures for the TP-fun routines

The TP functions are used to enter model parameters and other functions that depend on T and P. They have a limited sytax as the software must be able to calculate their first and second derivatives with respect to T and P.

The expressions are stored only once but the calculated values of the functions are stored separately for each equilibria as each equilibria can have different values of T and P.

4.1 Structure to store expressions of TP functions

In this record the coefficients and powers of T and P and possible unary functions used is stored. There is a simple parser that can read a simple TP function similar to what TC has. My idea has been to extend this a bit by allowing parenthesis in a more flexible way but that is not yet implemented. A TP function can refer to another TP function using the link array. This means a recursive evaluation of TP functions must be implemented.

This information must not be changed during parallel processing. During assessments one may calculate each experimental equilibria in parallel and as the coefficients of TP functions are varied by the assessment procedure one may think that this array must also be separate for each parallel process. But changes of coefficients are not done during the parallel execution so there is no problem.

```
  integer, parameter :: tpfun_expression_version=1
  TYPE tpfun_expression
! Coefficients, T and P powers, unary functions and links to other functions
     integer noofcoeffs,nextfrex
     double precision, dimension(:), pointer :: coeffs
! each coefficient kan have powers of T and P/V and links to other TPFUNS
! and be multiplied with a following LOG or EXP term.
     integer, dimension(:), pointer :: tpow
     integer, dimension(:), pointer :: ppow
     integer, dimension(:), pointer :: wpow
     integer, dimension(:), pointer :: plevel
     integer, dimension(:), pointer :: link
  END TYPE tpfun_expression
! These records are allocated when needed, not stored in arrays
```

4.2 Bits used in TP function status

These bits are used for different purposes

```
! BITS in TPFUN
! TPCONST     set if a constant value
! TPOPTCON    set if optimizing value
! TPNOTENT    set if referenced but not entered (when reading TDB files)
! TPVALUE     set if evaluated only explicitly (keeping its value)
  integer, parameter :: &
       TPCONST=0,    TPOPTCON=1,   TPNOTENT=2,    TPVALUE=3
```

4.3 Function root record type

In this record a name of the function is stored and the number of temperature ranges. For each range a low temperature limit and a link to a tpfun_expression record is stored. There is also a high temperature limit. As the number of ranges can vary the arrays are allocatable (declared as pointers). At the same time as a root record is reserved to store a TP function one also reserves a tpfun_parres record and in this the last calculated result of the function is stored. Saving the last calculated values speeds up calculation because the same function is often used in many parameters and needed many times for the same values of T and P. But in parallel processing the values of T and P can be different in each processor and thus the results must be separate in each.

```
  integer, parameter :: tpfun_root_version=1
  TYPE tpfun_root
! Root of a TP function including name with links to coefficients and codes
! and results.  Note that during calculations which can be parallelized
! the results can be different for each parallel process
     character*(lenfnsym) symbol
! limits are the low temperature limit for each range
! funlinks links to expression records for each range
! each range can have its own function, status indicate if T and P or T and V
! forcenewcalc force new calculation when optimizing variable changed
     integer noofranges,nextfree,status,forcenewcalc
     double precision, dimension(:), pointer :: limits
     TYPE(tpfun_expression), dimension(:), pointer :: funlinks
     double precision hightlimit
  END TYPE tpfun_root
! These records are stored in arrays as the actual function is global but each
! equilibrium has its own result array (tpfun_parres) depending on the local
! values of T and P/V.  The same indiex is used in the global and local arrays.
! allocated in init_gtp
  TYPE(tpfun_root), private, dimension(:), pointer :: tpfuns
```

4.4 Structure for calculated results of TP functions

This record contains the last values of T and P used for calculation the TP function and the results including first and second derivatives of T and P. These are ordered as F, F.T, F.P, F.T.T, F.T.P and F.P.P. The reason to have this as a separate array is that in parallel processing the values of T and P may be different in each process and thus the results also, whereas the values of the coefficients are the same. The size of all the function records are allocated dynamically by the tpfun_init subroutine.

```
  integer, parameter :: tpfun_parres_version=1
  TYPE tpfun_parres
! Contains a TP results, 6 double for results and 2 doubles for T and P
! values used to calculate the results
! Note that during calculations which can be parallelized the final
! results can be different for each parallel process
     integer forcenewcalc
     double precision, dimension(2) :: tpused
     double precision, dimension(6) :: results
  END TYPE tpfun_parres
! This array is local to the gtp_equilibrium_data record
! index the same as the function
```

5 Basic thermodynamic data structures

Below the data types in GTP will be explained and in some cases also how they are declared as arrays or inside other data types. Many of the declared variables are "private" which means they cannot be changed directly from outside the module. For such data that the user will manipulate there are subroutines or functions provided.

5.1 Global data

Information that is valid for the whole system is stored here. Note that T and P may be different in different parallel processes and if there is a prescribed value that is stored in a condition record. Maybe this is not really needed.

```
  TYPE gtp_global_data
! status should contain bits how advanced the user is and other defaults
! it also contain bits if new data can be entered (if more than one equilib)
     integer status
     character name*24
     double precision rgas,rgasuser,pnorm
  END TYPE gtp_global_data
  TYPE(gtp_global_data) :: globaldata
```

5.2 Version identification of the data structure

As the data structure is always open to changes and as such changes may affect other parts of the software most of the TYPE definitions have a "version" value. Whenever a change is made in a TYPE definition this version value should be incremented. The version number can be tested in other parts of the software, in particular when writing the data structure on a file and at a later time reading it back from a file. If the version number is not the same it may not be possible to read the file or the software must have provisions for reading records with different version numbers.

5.3 The ELEMENT data type

Elements are the building blocks of other data. They have a symbol, mass, reference state and some other values defined.

The elements in a system are stored in an array ELLISTA of these records in the order they are entered. Another integer array ELEMENTS have the elements in alphabetical order. These arrays are allocated in the subroutine init_gtp.

There are two predefined elements, the electron with symbol "/-" and the vacancy with symbol "Va". They have the indices -1 and 0. All real elements have numbers from 1 and higher.

The OC software is case insensitive, UPPER and lower case letters are treated identically. Thus Va, va and VA is the same. Elements with a single letter symbol must be followed by a space or a non-alphabetic character, for example a stoichiometric factor, to separate it from a following letter.

The advantage with ELLISTA is that the element is stored at an index which never change, it does not change when other elements are entered which would change the alphabetical order. Note that there is a cross index stored in ellista so it is possible to know the alphabetical order of the element.

It also means there are 3 ways to specify an element, its symbol, its alphabetical index (its index in ELEMENTS) and its index in ELLISTA. The ELLISTA index is most important and it will never change whereas the ELEMENTS index may change when new elements are entered that may change the alphabetical order. The link to an element from a species are always the ELLISTA.

In the species record the stoichiometric formula is stored by giving the index of the element in ellista. As this never change no rearrangement is needed if new elements are added.

The elements array is useful to make alphabetically ordered listing of data. Elements are normally never deleted but may be suspended i.e. hidden from application programs (this cannot be allowed during parallel processing).

```
! this constant must be incremented whenever a change is made in gtp_element
  INTEGER, parameter :: gtp_element_version=1
  TYPE gtp_element
! data for each element: symbol, name, reference state, mass, h298-h0, s298
     character :: symbol*2,name*12,ref_state*24
     double precision :: mass,h298_h0,s298
! splink: index of corresponding species in array splink
! Status bits are stored in the integer status
! alphaindex: the alphabetical order of this elements
! refstatesymbol: indicates H0 (1), H298 (0, default) or G (2) for endmembers
     integer :: splink,status,alphaindex,refstatesymbol
  END TYPE gtp_element
! allocated in init_gtp
  TYPE(gtp_element), private, allocatable :: ellista(:)
  INTEGER, private, allocatable :: ELEMENTS(:)
```

5.4 Species

Here some of the useful features of Fortran08 data structuring appears and can be explained. The species have a name, mass and charge as fixed attributes. It also has a status word and a link back to the array where the species are arranged in alphabetical order. It has an integer giving the number of elements in the species (must be larger than zero). Finally there are two arrays which have no size specified in the declaration, these must be allocated when the species is entered. The reason to have arrays that can be allocated is that some species has just one element, other may have two, three or more. The dimension of these arrays are stored in noofel (It can actually also be obtained by the Fortran08 build-in function SIZE). The indices stored in ellinks are the location, i.e. index to ellista. This does not change even if the alphabetical order of elements is changed.

For saving and reading a stored datastructure from a file it is necessary to have the number of elements as a stored integer because this is needed to allocate space for these arrays before reading the content of these arrays from the file. The SIZE function is useless in that case.

One can declare arrays that should be allocated dynamically either as ALLOCATABLE or POINTER. Note that allocating POINTER variables is a likely source of memory leaks.

The species record is an interesting example of an array of structures that in itself contains allocatable arrays that can vary in size depending on the data stored there. This feature will be utilized extensively when storing data for phases.

It is also a very interesting feature to use indices to other arrays as links rather than pointers which is necessary in most other languages with data structures. There

are pointers also in Fortran08 but pointers have a bad habit of being confusing and complicated to use, one is never sure if one has the address of a pointer or the content (which is also a memory address).

The integer array "species" has the species in alphabetical order by providing the correct index to the SPLISTA.

```
! this constant must be incremented whenever a change is made in gtp_species
  INTEGER, parameter :: gtp_species_version=1
  TYPE gtp_species
! data for each species: symnol, mass, charge, status
! mass is in principle redundant as calculated from element mass
     character :: symbol*24
     double precision :: mass,charge
! alphaindex: the alphabetical order of this species
! noofel: number of elements
     integer :: noofel,status,alphaindex
! Use an integer array ellinks to indicate the elements in the species
! The corresponing stoichiometry is in the array stochiometry
! ???? these should not be pointers, changed to allocatable ????
     integer, dimension(:), allocatable :: ellinks
     double precision, dimension(:), allocatable :: stoichiometry
  END TYPE gtp_species
! allocated in init_gtp
  TYPE(gtp_species), private, allocatable :: splista(:)
  INTEGER, private, allocatable :: SPECIES(:)
```

5.5 Components

It will be possible to define different components for each equilibrium. A component must be a species. Initially the species identical to the elements are the components. The components are part of the "gtp_equilibrium_data" record and after a calculation the chemical potentials are stored in these records.

```
! this constant must be incremented whenever a change is made in gtp_component
  INTEGER, parameter :: gtp_component_version=1
  TYPE gtp_components
! The components are simply an array of indices to species records
! the components must be "orthogonal".  There is always a "systems components"
! that by default is the elements.
! Later one may implement that the user can define a different "system set"
! and also specific sets for each phase.
! The reference state is set as a phase and value of T and P.
! The name of the phase and its link and the link to the constituent is stored
```

```
! the endmember array is for the reference phase to calculate GREF
! The last calculated values of the chemical potentials (for user defined
! and default reference states) should be stored here.
! molat is the number of moles of components in the defined reference state
     integer :: splink,phlink,status
     character*16 :: refstate
     integer, dimension(:), allocatable :: endmember
     double precision, dimension(2) :: tpref
     double precision, dimension(2) :: chempot
     double precision mass,molat
  END TYPE gtp_components
! allocated in gtp_equilibrium_data
```

5.6 Phase datatypes

The data stored for a phase is very complex. The data are all accessed from a phase "root" record (except when save/read from a file). The phase has a phase_varres record which is stored in the gtp_equilibrium_data record that keeps the last calculated results as this will be different in each different parallel process. In the phase_varres record all fraction variables (and some additional information) that can be different for each parallel process is also stored. If the phase has two or more composition sets each of these has its own phase_varres record. The phase_varres records are an array inside the gtp_equilibrium_data record and each phase store the links to its composition sets by indices to this array. This is necessary as one can have several gtp_equilibrium_data records and one must have the same number of composition sets for a phase in all. Whenever new composition sets are created in parallel computing it must be done for all threads at the same time.

The root record of a phase has data and links to all necessary information listed below:

- The endmember record list. There can be two endmember lists, one for each fraction types.
- The phase_varres record representing a composition set. Each each phase kan have up to 9 composition sets. The phase_varres records are declared as an array in the gtp_equilibrium_data record because ech equilibrium must have a unique set of fractions and results. In this record there are also some status bits that may be different in different parallel processes like if the phase (composition set) is FIXED/ENTERED/etc.

  A phase may exist simultaneously with different constituent fractions like in a miscibility gap or when a phase can order. To identify a composition set one can use the number symbol "#" follwed by a digit. When the composition set is created one can also add a pre- and suffix.

In the phase_varres record there are also arrays to store all calculated G and its first and second derivatives (also derivatives with respect to fractions). Calculated values of other properties like TC, V, MQ etc (and their derivatives) are also stored if there are parameters for these properties.

For the ionic liquid model the number of sites depend on the constitution and thus there are also provisions to have the number of sites on each sublattice in this record.

- The addition list. This is a pointer to an addition record (there can be several linked sequentially). The only current addition record is used to specify the magnetic magnetic contribution and for other additions new subroutines must be written to enter, list and calculate the additions. See section 5.6.9.

These records may in turn point to other records, as already mentioned the endmember record has links to another endmember records and to an interaction records and both of these records has a link to a list of property records where there are links to TP functions. Some of these links are indexes to arrays of these records but for example the parameters stored in endmember, interactions and property records are created dynamically and can only be accessed by pointers. The property records have links to TP functions but these links are indices to the array of TP function structure. The reason to have an array for these is that there are separate arrays with the calculated results of the TP functions in the gtp_equilibrium_data record as these must be local to each equilibrium, whereas the function expressions are the same for all equilibria.

5.6.1 Permutations of constituents

A complication for endmembers and interaction records is to handle permutations of constituents when one has ordering. It is clumsy to store identical permutation of constituents (like A:A:A:B, A:A:B:A, A:B:A:A, B:A:A:A for a 4 sublattice FCC ordering) in separate endmembers but one should have a single endmember for all permutations. It is necessary that the software can handle such permutations for multicomponent systems, otherwise the databases must contain several 1000 permutations of identical values of the same parameter. A 4 component FCC ordered system with 4 sublattices have totally 256 endmembers but only 35 unique ones.

The permutations create complications also for the interaction records which depend on the order of constituents in the endmember. An interaction records for an interaction between A and B belonging to an end member A:A:A:A (which has no permutations) but there are 4 permutations of the interaction record (B can be in any of the 4 sublattices). So each combination must be considered.

The second order interaction A,B:A,C:A:A this has 3 additional permutations compared to the first order as the interaction with C can be placed in any of the 3 remaining

sublattices. But the second order interaction A,B:A,B:A:A has a more complicated permutations. If the first interaction with B is in the first sublattice, the second interaction with B can be placed in any or the remaining 3 but if the first level interaction in B is in the second sublattice the second level interaction can only be in the third or forth. In Table 3 some permuations are shown.

Table 3: Possible permutations of some endmembers and interaction records.

endmember	1st level interaction	2nd level interaction
A: A: A: A	A,B: A: A: A	A,B: A,B: A: A
		A,B: A: A,B: A
		A,B: A: A: A,B
	A: A,B: A: A	A: A,B: A,B: A
		A: A.B: A: A,B
	A: A: A,B: A	A: A: A,B: A,B
	A: A: A: A,B	no permutation
A: A: A: B	A,B: A: A: B	A,B: A,B: A: B
		A,B: A: A,B: B
	A: A,B: A: B	A: A,B: A,B: B
	A: A: A,B: B	no permutation
A: A: B: A	A,B: A: B: A	A,B: A,B: B: A
		A,B: A: B: A,B
	A: A,B: B: A	A: A,B: B: A,B
	A: A: B: A,B	no permutation
etc.		

Reciprocal parameters like L(fcc,A,B:A,B:A:A) are important to approximate the short range order contibutions to the Gibbs energy.

The permutations means some complications when entering parameters as links to the fractions of all permutations are created at that stage. It also means that all constituents must be ordered, always in alphabetical order, so it is possible to find a parameter if its value should be changed. When calculating with the permutations the data structure created when entering the parameter must contain information on the number of permutations inherited from the endmember up to the second level interaction. For permutations no interactions higher than the 2nd order are allowed.

In the first release of OC most of the permutations for the tetrahedron 4 sublattice fcc (and hcp) ordering has been implemented. The 4 sublattice bcc permutations are more complex and will be implemented later. All endmember permutations and all first level level have been implemented but only a limted set of 2nd order interactions, i.e. all for binary and ternary system.

An ordered phase will in many cases have a disordered fraction set for the disordered state and there will be parameters which depend on these fractions. There can be endmembers and interaction parameters for each fraction set. The parameters for each fraction set will be evaluated separately and added together at the end, taking into account the chain rule for the derivatives.

5.6.2 Property types for different fraction sets

An endmember or interaction can have several property records, normally one for the Gibbs energy but also for Curie temperature, mobilities etc. But there can be endmembers and interaction records without property records just because some higher interaction record have a property record.

For mobilities one can use a special type of property that is specific to a constituent like MQ&FE. In a property record one must have an adjustable array for function links, for example when there are several RK terms, see 2.3.2. These links can be allocated dynamically and if there are new RK terms added or deleted this array can be extended or decreased dynamically. For each property record one can store a reference to the origin of the data.

I am not sure exactly how one should in listings identify parameters belonging to different fraction sets. In a SIGMA phase with 3 sublattices one will have endmember parameters for some like G(SIGMA,FE:CR:FE), which are multiplied with the site fractions of these constituents, $y_{\rm Fe}^{(1)}y_{\rm Cr}^{(2)}y_{\rm Fe}^{(3)}$. But we will also have a fraction set which in this case will be identical to the mole fractions of Fe and Cr,

$x_{\rm Fe} = (10y_{\rm Fe}^{(1)} + 4y_{\rm Fe}^{(2)} + 16y_{\rm Fe}^{(3)})/30$

An endmember parameter for this fraction set is currently specified as GD(SIGMA,FE) and an interaction as GD(SIGMA,CR,FE). If one considers extending to several levels of fraction sets one should maybe use an integer specification like G#2(SIGMA,FE) etc. In any cases a property without specification will belong to the primary site fraction set which is the major one, all other fractions are calculated from these fractions.

For FCC ordering with interstitials the second fraction set will not be mole fractions but the sum of the site fractions of the substitutional sublattices

$z_{Fe} = 0.25\sum_{s=1,4} y_{\rm Fe}^{(s)}$

5.6.3 Selected_element_reference

With Thermo-Calc it is has been a little awkward that one may not read the normal reference phase for an element from the database. For example the gas phase may not be included for calculations with nitrogen present but one would like to have partial pressures or activities of nitrogen referenced to the gas.

An attempt to handle this has been made by entering by default a phase called "SELECT_ELEMENT_REFERENCE". This phase is always hidden and cannot be included in any calculation. Its location and index is 0 (zero) so it does not show up in the number of phases. But the user can list the data of this phase and also amend them. The idea is that a database will automatically enter the appropriate data for each element in this phase and in calculations this phase will be the default reference state for activities or enthalpies (at current temperature). One must have a phase and not just a TP function to handle elements with magnetic transitions. This reference phase is in fact a different phase for each element, one must be able to use different magnetic functions for different elements. All details for this are not worked out and it may not even be a very good idea.

5.6.4 The endmember record

As already described the phase record is a root of two lists of endmembers, one for each of the two possible fraction sets. The endmember records are allocated by pointers when needed and contains several links to other records:

- one link to the next endmember record,
- one link to an interaction record, the root of a binary tree,
- one link to a list of property records and
- links to one constituent in each sublattice

All the links except those to the constituents may be empty. The links to the constituents are not real links but an integer index of the array of constituents stored in the phase record (described later). The endmembers are always arranged in increasing value of these indices, in the order of the sublattices. The endmember properties are multiplied with the fractions of these constituents when a calculation is performed.

When the ordering options has been implemented, see 5.6.1, an endmember may have several permutations of the constituent indices. That is the reason to have two-dimensional arrays for the constituent indices.

One may have endmembers that does not depend on the constituent in a specific sublattice and this is represented by the fraction index for this sublattice is negative, usually -99. This is called a "wildcard" and represented by an asterix "*" when entering or listing the parameter.

```
! this constant must be incremented whenever a change is made in gtp_endmember
  INTEGER, parameter :: gtp_endmember_version=1
  TYPE gtp_endmember
```

```
! end member parameter record, note ordered phases can have
! several permutations of fraction pointers like for B2: (Al:Fe) and (Fe:Al).
! There are links (i.e. indices) to next end member and to the interactio tree
! and to a list of property record
! The phase link is needed for SAVE/READ as one cannot know the number of
! sublattices otherwise.  One could just store nsl but a link back to the
! phase record might be useful in other cases.
! noofpermut: number of permutations (for ordered phases: (Al:Fe) and (Fe:Al)
! phaselink: index of phase record
! antalem: sequenial order of creation, useful to keep track of structure
! propointer: link to properties for this endmember
! nextem: link to next endmember
! intponter: root of interaction tree of parameters
! fraclinks: indices of fractions to be multiplied with the parameter
     integer :: noofpermut,phaselink,antalem
     TYPE(gtp_property), pointer :: propointer
     TYPE(gtp_endmember), pointer :: nextem
     TYPE(gtp_interaction), pointer :: intpointer
! there is at least one fraclinks per sublattice
! the second index of fraclinks is the permutation (normally only one)
! the first indec of fraclinks points to a fraction for each sublattice.
! The fractions are numbered sequentially independent of sublattices, a
! sigma phase with (FE:CR,MO:CR,FE,MO) has 6 fractions (incl one for FE in
! first sublattice) and the end member (FE:MO:CR) has the fraclinks 1,3,4
! This means these values can be used as index to the array with fractions.
! The actual species can be found via the sublattice record
!    integer, dimension(:,:), pointer :: fraclinks
     integer, dimension(:,:), allocatable :: fraclinks
  END TYPE gtp_endmember
! dynamically allocated when entering a parameter
```

5.6.5 The interaction record

The interaction records forms a binary tree starting from an endmember record. It thus has two pointers to other interaction records, one which is the *next* on the same interaction level, excluding the interacting constituent in the current record, and one to a *higher* level, including the current interacting constituent. In this way any level of interaction parameter can be defined. However, only the first two levels (binary and ternary) can be composition dependent, see 2.3.2.

There is also a link to a list of property records. This link can be empty if there is a link to a higher order interaction.

The interaction record specifies one additional constituent in a sublattice. The remaining constituents are given by the endmember record and any lower interaction

records with a higher link to this one. Again one can have permutations of this interacting constituent in the different sublattices, see section 5.6.1.

The interaction records are always linked from the first possible endmember, i.e. that with the lowest set of constituent indices in all sublattices. If there are several levels of interactions the lowest interaction has the lowest constituent index. There is no ordering of interacting constituents on the same level.

```
! this constant must be incremented when a change is made in gtp_interaction
  INTEGER, parameter :: gtp_interaction_version=1
  TYPE gtp_interaction
! this record constitutes the parameter tree. There are links to NEXT
! interaction on the same level (i.e. replace current fraction) and
! to HIGHER interactions (i.e. includes current interaction)
! There can be several permutations of the interactions (both sublattice
! and fraction permuted, like interaction in B2 (Al:Al,Fe) and (Al,Fe:Al))
! The number of permutations of interactions can be the same, more or fewer
! comparared to the lower order parameter (endmember or other interaction).
! The necessary information is stored in noofip.  It is not easy to keep
! track of permutations during calculations, the smart way to store the last
! permutation calculated is in this record ... but that will not work for
! parallel calculations as this record is static ...
! status: may be useful eventually
! antalint: sequential number of interaction record, to follow the structure
! order: for permutations one must have a sequential number in each node
! propointer: link to properties for this parameter
! nextlink: link to interaction on same level (replace interaction)
! highlink: link to interaction on higher level (include this interaction)
! sublattice: (array of) sublattices with interaction fraction
! fraclink: (array of) index of fraction to be multiplied with this parameter
! noofip: (array of) number of permutations, see above.
     integer status,antalint,order
     TYPE(gtp_property), pointer :: propointer
     TYPE(gtp_interaction), pointer :: nextlink,highlink
     integer, dimension(:), allocatable :: sublattice,fraclink,noofip
  END TYPE gtp_interaction
! allocated dynamically and linked from endmember records
```

5.6.6 The property record

The endmembers and interaction records is the start of a list of property records. A property list can be empty if an endmember or interaction record is needed to specify some higher interaction.

There is a property index, see 2.3.5 and a *nextpr* link to another property record. One can have a link to a reference for the parameter (published paper or similar), see 5.6.7.

The actual value of the property is stored as an index to a TP function. These functions are stored in array and cannot be deleted (but they can be changed) so the index is a stable link. The calculated values of a property is stored in another array, local to the equilibrium record, see 6.2.5, as one may have different values in each thread in parallel processing or when running assessments, as these use different equilibrium records.

For binary and ternary interactions one can have a degree larger than zero which means there are several TP functions linked to this property. See 2.3.2.

```
! this constant must be incremented when a change is made in gtp_property
  INTEGER, parameter :: gtp_property_version=1
  TYPE gtp_property
! This is the property record.  The end member and interaction records
! have pointer to this.  Severall different properties can be linked
! from a parameter record like G, TC, BMAGN, VA, MQ etc.
! Some properties are connected to a constituent (or component?) like the
! mobility and also the Bohr mangneton number.
! Allocated as linked from endmembers and interaction records
! reference: can be used to indicate the source of the data
! refix: can be used to indicate the source of the data
! nextpr: link to next property record
! extra: TOOP and KOHLER can be implemented inside the property record
! proptype: type of propery, 1 is G, other see parameter property
! degree: if parameter has Redlich-Kister or similar degrees (powers)
! degreelink: indices of TP functions for different degrees (0-9)
! protect: can be used to prevent listing of the parameter
! antalprop: probably redundant (from the time of arrays of propery records)
     character*16 reference
     TYPE(gtp_property), pointer :: nextpr
     integer proptype,degree,extra,protect,refix,antalprop
     integer, dimension(:), allocatable :: degreelink
  END TYPE gtp_property
! property records, linked from endmember and interaction records, allocated
! when needed.  Each propery like G, TC, has a property record linking
! a TPFUN record (by index to tpfun_parres)
```

5.6.7 Bibliographic references

It is important to document the source of the data and each property can have a unique bibliographic reference stored in this record. It would typically be a published paper or some internal written documentation.

```
! this constant must be incremented when a change is made in gtp_biblioref
! old name gtp_datareference
```

```
  INTEGER, parameter :: gtp_biblioref_version=1
  TYPE gtp_biblioref
! store data references
! reference: can be used for search of reference
! refspec: free text
     character*16 reference
     character*64, dimension(:), allocatable :: refspec
  END TYPE gtp_biblioref
! allocated in init_gtp
  TYPE(gtp_biblioref), private, allocatable :: bibrefs(:)
```

5.6.8 Parameter property identification

The propery list has an index and the meaning of this index is given the gtp_propid record. A few of these are predefined but a programmer can add more, see section 2.3.3. For any added property software must also be written to handle such properties during calculations. The implemented properties can be listed, see 8.8.26 and the values of these properties can be listed similarly like for state variables, see 8.8.27.

```
! this constant must be incremented when a change is made in gtp_propid
  INTEGER, parameter :: gtp_propid_version=1
  TYPE gtp_propid
! this identifies different properties that can depend on composition
! Property 1 is the Gibbs energy and the others are usually used in
! some function to contribute to the Gibbs energy like TC or BMAGN
! But one can also have properties used for other things like mobilities
! with additional especification like MQ&FE
! symbol: property identifier like G for Gibbs energy
! note: short description for listings
! prop_elsymb: additional for element dependent properties like mobilities
     character symbol*4,note*16,prop_elsymb*2
! Each property has a unique value of idprop.  Status can state if a property
! has a constituent specifier or if it can depend on T or P
     integer status
! this can be a constituent specification for Bohr mangetons or mobilities
! such specification is stored in the property record, not here
!    integer prop_spec,listid
! >>> added "listid" as a conection to the "state variable" listing here.
! This replaces TC, BMAG, MQ etc included as "state variables" in order to
! list their values.  In this way all propids become available
  end TYPE gtp_propid
! the value TYPTY stored in property records is "idprop" or
! if IDELSUFFIX set then 100*"idprop"+ellista index of element
! if IDCONSUFFIX set then 100*"idprop"+constituent index
! When the parameter is read the suffix symbol is translated to the
```

```
! current element or constituent index
  TYPE(gtp_propid), dimension(:), private, allocatable :: propid
```

5.6.9 Additions to the Gibbs energy

Different contributions to the Gibbs energy can be specified in the software using the data structure below. Such additions normally depend on a number of composition dependent properties like the Curie temperature, the Bohr magneton number, the Debye temperature etc. These can be added as properties and entered as endmember and interaction parameters with a unique index and symbol see 5.6.8.

When implementing a new addition subroutines must be written for entering, listing and calculating the property, including analytical first and second derivatives with respect to T, P and the constitution.

One may also add properties that does not contribute to the Gibbs energy but depend on the constitution of the phase, like mobilities, viscosities, resistivity etc. Such properties are calculated together with the Gibbs energy and their values can be extracted by appropriate subroutines.

```
! this constant must be incremented when a change is made in gtp_phase_add
  INTEGER, parameter :: gtp_phase_add_version=1
  TYPE gtp_phase_add
! record for additions to the Gibbs energy for a phase like magnetism
! addrecno: ?
! aff: antiferomagnetic factor (Inden model)
! need_property: depend on these properties (like Curie T)
! explink: function to calculate with the properties it need
! nextadd: link to another addition
     integer type,addrecno,aff
     integer, dimension(:), allocatable :: need_property
     TYPE(tpfun_expression), dimension(:), pointer :: explink
     TYPE(gtp_phase_add), pointer :: nextadd
     type(gtp_elastic_modela), pointer :: elastica
  END TYPE gtp_phase_add
! allocated when needed and linked from phase record
```

5.6.10 The elastic model

This is a tentative record to model elastic contributions to the Gibbs energy.

```
! addition record to calculate the elastic energy contribution
! declared as allocatable in gtp_phase_add
! this constant must be incremented when a change is made in gtp_elastic_modela
```

```
  INTEGER, parameter :: gtp_elastic_modela_version=1
  TYPE gtp_elastic_modela
! lattice parameters (configuration) in 3 dimensions
     double precision, dimension(3,3) :: latticepar
! epsilon in Voigt notation
     double precision, dimension(6) :: epsa
! elastic constant matrix in Voigt notation
     double precision, dimension(6,6) :: cmat
! calculated elastic energy addition (with derivative to T and P?)
     double precision, dimension(6) :: eeadd
! maybe more
  end TYPE gtp_elastic_modela
```

5.6.11 The phase and composition set indices

In many cases one must specify both a phase and a composition set and this is done by separate indices in most subroutine calls. As this is not very convenient an attempt to introduce a single index for both phase and composition sets has been med by the "phase tuple" type. This is a simple record with five integers, ixphase is the index of the phase in alphabetical order, compset the composition set index, phaseix is the index of the phase in the phlista array, lokvares is the index of the phase_varres record which is useful to retrieve data. If there is a higher composition set for the same phase then nextcs is the phase tuple index to the phase, otherwise nextcs is zero. Ixphase is probably redundant. There is an array with the phasetuple records called PHASETUPLE which is updated whenever a composition set is created or deleted.

```
! this constant must be incremented when a change is made in gtp_phasetuple
  INTEGER, parameter :: gtp_phasetuple_version=2
  TYPE gtp_phasetuple
! for handling a single array with phases and composition sets
! first index is phase index, second index is composition set
! ADDED also index in phlista (ixphase) and phase_varres (lokvares) and
! nextcs which is nonzero if there is a higher composition set of the phase
! A tuplet index always refer to the same phase+compset.  New tuples with
! the same phase and other compsets are added at the end.
     integer phaseix,compset,ixphase,lokvares,nextcs
  end TYPE gtp_phasetuple
```

In some records and subroutine calls the phase tuple is used to specify both phase and composition sets, in others separate integer indices are used.

If each phase has just a single composition set the phase tuple index is the same as the phase index. Additional composition sets of the same phase with have higher indices than the last phase index.

5.6.12 The phase record

We have finally reached the phase record itself. As this refers to many of the structures above it is declared in the source code after all of them and to simplify the updating of this documentation we follow the order of the declarations in the source code. It is complicated enough already ...

The phase record had originally two parts but these have been merged to a single record. The link to the composition set data (the phase_varres record) has also recently been changed so the link to all of them are stored in the phase record as an array of indices to the array of phase_varres records in the equilibrium record. The phase_varres record contains all *dynamic* data that change during iterations and has also all calculated results. The endmember and interaction records contain the *static* data that does not change (except the calculated values of all TP functions9 but this is also stored the equilibrium record in the tp_res array.

When a phase is created a link to a record in the phase_varres array, i.e. an integer index, is stored in the array linktocs(1). If additional composition sets are created they are stored sequentially in the same array. A phase cannot have more than 9 composition sets.

```
! a smart way to have an array of pointers used in gtp_phase
  TYPE endmemrecarray
     type(gtp_endmember), pointer :: p1
  end TYPE endmemrecarray
!-------------------------------------------------------------
! this constant must be incremented when a change is made in gtp_phase
  INTEGER, parameter :: gtp_phase_version=1
  TYPE gtp_phaserecord
! this is the record for phase model data. It points to many other records.
! Phases are stored in order of creation in phlista(i) and can be found
! in alphabetical order through the array phases(i)
! sublista is now removed and all data included in phlista
! sublattice and constituent data (they should be merged)
! The constituent link is the index to the splista(i), same function
! as LOKSP in iws.  Species in alphabetcal order is in species(i)
! One can allocate a dynamic array for the constituent list, done
! by subroutine create_constitlist.
! Note that the phase has a dynamic status word status2 in gtp_phase_varres
! which can be differnt in different parallel calculations.
! This status word has the FIX/ENT/SUS/DORM status bits for example
! name: phase name, note composition sets can have pre and suffixes
! model: free text
! phletter: G for gas, L for liquid
! alphaindex: the alphabetcal order of the phase (gas and liquids first)
     character name*24,models*72,phletter*1
```

```
      integer status1,alphaindex
! noofcs: number of composition sets,
! nooffs: number of fraction sets (replaces partitioned phases in TC)
      integer noofcs,nooffs
! additions: link to addition record list
! ordered: link to endmember record list
! disordered: link to endmember list for disordered fractions (if any)
      TYPE(gtp_phase_add), pointer :: additions
      TYPE(gtp_endmember), pointer :: ordered,disordered
! To allow parallel processing of endmembers, store a pointer to each here
      integer noemr,ndemr
      TYPE(endmemrecarray), dimension(:), allocatable :: oendmemarr,dendmemarr
! noofsubl: number if sublattices
! tnooffr: total number of fractions (constituents)
! linktocs: array with indices to phase_varres records
! nooffr: array with number of constituents in each sublattice
! Note that sites are stored in phase_varres as they may vary with the
! constituion for ionic liquid)
! constitlist: indices of species that are constituents (in all soblattices)
      integer noofsubl,tnooffr
      integer, dimension(9) :: linktocs
      integer, dimension(:), allocatable :: nooffr
! number of sites in phase_varres record as it can vary with composition
      integer, dimension(:), allocatable :: constitlist
! used in ionic liquid:
! i2slx(1) is index of Va, i2slx(2) is index if last anion (both can be zero)
      integer, dimension(2) :: i2slx
! allocated in init_gtp.
  END TYPE gtp_phaserecord
! NOTE phase with index 0 is the reference phase for the elements
! allocated in init_gtp
  TYPE(gtp_phaserecord), private, allocatable :: phlista(:)
  INTEGER, private, allocatable :: PHASES(:)
```

5.7 State variables and the state variable record

Conditions and results are obtained as values of state variables. There are many of these like $G, H, T, x(<\text{component}>)$ etc. They are stored in the software as a record of the type below where istv is an integer giving the basic type and indices can give additional specification. Some properties like chemical potentials can have a reference state and one can also define a unit like Kelvin or calories. At present this is used only to specify if a composition variable is a fraction or a percent.

```
! this constant must be incremented when a change is made in gtp_state_variable
```

```
  INTEGER, parameter :: gtp_state_variable_version=2
  TYPE gtp_state_variable
! this is to specify a formal or real argument to a function of state variables
! statev/istv: state variable index
! phref/iref: if a specified reference state (for chemical potentials)
! unit/iunit: 100 for percent, no other defined at present
! argtyp together with the next 4 integers represent the indices(4), only 0-4
! argtyp=0: no indices (T or P)
! argtyp=1: component
! argtyp=2: phase and compset
! argtyp=3: phase and compset and component
! argtyp=4: phase and compset and constituent
! ?? what is norm ??
     integer statevarid,norm,unit,phref,argtyp
! these integers represent the previous indices(4)
     integer phase,compset,component,constituent
! a state variable can be part of an expression with coefficients
! the coefficient can be stored here.  Default value is unity.
! In many cases it is ignored
     double precision coeff
! NOTE this is also used to store a condition of a fix phase
! In such a case statev is negative and the absolute value of statev
! is the phase index.  The phase and compset indices are also stored in
! "phase" and "compset" ??
! This is a temporary storage of the old state variable identifier
     integer oldstv
  end TYPE gtp_state_variable
! used for state variables/properties in various subroutines
```

6 Use of the thermodynamic data structures

This section describes how the basic thermodynamic data structures are used in various applications like calculating the Gibbs energy, handling errors, step for property diagrams, mapping of phase diagram, assessments etc.

6.1 Error handling

There is a global error code defined as part of the defined in the TYPE gtp_parerr. This has just one integer variable called bmperr and there is one variable of this type called gx which is decleared THREADPRIVATE for use in parallel processing.

The error code is gx%bmperr in the whole OC system. Whenever there is an error gxraising the error and usually the subroutine is exited directly. The error code should

be tested after each subroutine call and maybe the calling routine can handle the error.

There are error messages defined for the errors generated by GTP.

6.2 Calculations

So far we have described how to handle the data for the phase. In order to calculate an equilibrium many values, like the constituent fractions, amounts of phase, temperature etc may change. As mentioned several times we must also be able to handle several separate equilibria, either as threads in parallel computing or as experimental data in assessments. Each of these separate datasets are stored in a gtp_equilibrium_data record.

6.2.1 Conditions

We must also handle conditions set by the user. Typically a condition is a state variable equal to a value like T=1273 or w(c)=0.01. At present simple assignments of state variables are the main types of conditions allowed but for phase compositions it is possible to use expressions like x(liq,fe)-x(bcc,fe)=0 to find a congruent and to set a state variables equal to a symbolic value like w(liq,b)=whatever.

```
! this constant must be incremented when a change is made in gtp_condition
! NOTE on unformatted SAVE files the conditions are written as texts
  INTEGER, parameter :: gtp_condition_version=1
  TYPE gtp_condition
! these records form a circular list linked from gtp_equilibrium_data records
! each record contains a condition to be used for calculation
! it is a state variable equation or a phase to be fixed
! The state variable is stored as an integer with indices
! NOTE: some state variables cannot be used as conditions: Q=18, DG=19, 25, 26
! There can be several terms in a condition (like x(liq,c)-x(fcc,c)=0)
! noofterms: number of terms in condition expression
! statev: the type of state variable (must be the same in all terms)
!           negative value of statev means phase index for fix phase
! active: zero if condition is active, nonzero for other cases
! unit: is 100 if value in percent, can also be used for temperature unit etc.
! nid: identification sequential number (in order of creation), redundant
! iref: part of the state variable (iref can be comp.set number)
! iunit: ? confused with unit?
! seqz is a sequential index of conditions, used for axis variables
! experimettype: inequality (< 0 or > 0) and/or percentage (-101, 100 or 101)
! symlink: index of symbol for prescribed value (1) and uncertainty (2)
! condcoeff: there is a coefficient and set of indices for each term
! prescribed: the prescribed value
```

```
! NOTE: if there is a symlink value that is the prescribed value
! current: the current value (not used?)
! uncertainty: the uncertainty (for experiments)
     integer :: noofterms,statev,active,iunit,nid,iref,seqz,experimenttype
!    TYPE(putfun_node), pointer :: symlink1,symlink2
! better to let condition symbol be index in svflista array
     integer symlink1,symlink2
     integer, dimension(:,:), allocatable :: indices
     double precision, dimension(:), allocatable :: condcoeff
     double precision prescribed, current, uncertainty
! currently this is not used but it will be
     TYPE(gtp_state_variable), dimension(:), allocatable :: statvar
     TYPE(gtp_condition), pointer :: next, previous
  end TYPE gtp_condition
! declared inside the gtp_equilibrium_data record
```

6.2.2 State variable functions

In this record the description of a state variable function is stored. The actual expression is stored using the PUTFUN subroutine in the metlib package. The calculated results of a state variable function is stored as a double precision array svfunres in the equilibrium data record. State variable function values are a single value, not 6 as for the TPfuns, as one cannot calculate a derivative with respect to anything.

The "dot" derivative expression available in Thermo-Calc has been implemented for one kind of calculations, $H.T$ meaning the partial derivative of enthalpy with respect to temperature, normally this is the heat capacity. Such a property can only be calculated in connection with an equilibrium calculation or a property diagram.

```
! this constant must be incremented when a change is made in gtp_putfun_lista
  INTEGER, parameter :: gtp_putfun_lista_version=1
  TYPE gtp_putfun_lista
! these are records for state variable functions.  The function itself
! is handelled by the putfun package.
! linkpnode: pointer to start node of putfun expression
! narg: number of symbols in the function
! nactarg: number of actual parameter specifications needed in call
!   (like @P, @C and @S
! status: can be used for various things
! status bit SVFVAL=0 means value evaluated only when called with mode=1
! SVCONST bit set if symbol is just a constant value (linknode is zero)
! eqnoval: used to specify the equilibrium the value should be taken from
!    (for handling what is called "variables" in TC)
! name: name of symbol
     integer narg,nactarg,status,eqnoval
```

```
     type(putfun_node), pointer :: linkpnode
     character name*16
     double precision value
! this array has identification of state variable (and other function) symbols
     integer, dimension(:,:), pointer :: formal_arguments
  end TYPE gtp_putfun_lista
! this is the global array with state variable functions
  TYPE(gtp_putfun_lista), dimension(:), allocatable :: svflista
! NOTE the value of a function is stored locally in each equilibrium record
! in array svfunres.
! The number of entered state variable functions. Used when a new one stored
  integer, private :: nsvfun
```

6.2.3 Fraction sets

A phase can have several composition sets, meaning that it can be stable with two or more different compositions. A phase can also have two fraction sets, explained in more detail in 5.6.2.

```
! this constant must be incremented when a change is made in gtp_fraction_set
  INTEGER, parameter :: gtp_fraction_set_version=1
  TYPE gtp_fraction_set
! info about disordred fractions for some phases like ordered fcc, sigma etc
! latd: the number of sublattices added to first disordred sublattice
! ndd: sublattices for this fraction set,
! tnoofxfr: number of disordered fractions
! tnoofyfr: same for ordered fractions (=same as in phlista).
! varreslink: index of disordered phase_varres,
! phdapointer: pointer to the same phase_varres record as varreslink
!    (Note that there is a bit set indicating that the sublattices should
!    be taken from this record)
! totdis: 0 indicates no total disorder (sigma), 1=fcc, bcc or hcp
! id: parameter suffix, D for disordered
! dsites: number of sites in sublattices, disordred fractions stored in
!    another phase_varres record linked from phdapointer
! splink: pointers to species record for the constituents
! nooffr: the number of fractions in each sublattice
! y2x: the conversion from sublattice constituents to disordered and
! dxidyj: are the the coeff to multiply the y fractions to get the disordered
!        xfra(y2x(i))=xfra(y2x(i))+dxidyj(i)*yfra(i)
! disordered fractions stored in the phase_varres record with index varreslink
!    (also pointed to by phdapointer).  Maybe phdapointer is redundant??
! arrays originally declared as pointers now changed to allocatable
     integer latd,ndd,tnoofxfr,tnoofyfr,varreslink,totdis
     character*1 id
```

```
     double precision, dimension(:), allocatable :: dsites
     integer, dimension(:), allocatable :: nooffr
     integer, dimension(:), allocatable :: splink
     integer, dimension(:), allocatable :: y2x
     double precision, dimension(:), allocatable :: dxidyj
! factor needed when reading from TDB file for sigma etc.
     double precision fsites
! in parallel processing the disordered phase_varres record is linked
! by this pointer,  used in parcalcg and calcg_internal
     TYPE(gtp_phase_varres), pointer :: phdapointer
  END TYPE gtp_fraction_set
! these records are declared in the phase_varres record as DISFRA for
! each composition set and linked from the phase_varres record
```

6.2.4 The phase_varres record for composition sets

Each composition set of a phase has a record as described below. It contains all data that can vary during calculations like the constituent fractions, the amount of the phase, etc. It also contains all results from a calculation. An array phase_varres is allocated in the gtp_equilibrium_record for this purpose and in the phase record there are indices to the phase_varres records for its composition sets.

As composition sets can be created and deleted there is a free list maintained using the integer *nextfree*. The first free phase_varres record is given by the global variable csfree. This list is maintained in the first equilibrium record, which is pointed to by the global variable FIRSTEQ.

```
! this constant must be incremented when a change is made in gtp_phase_varres
  INTEGER, parameter :: gtp_phase_varres_version=1
  TYPE gtp_phase_varres
! Data here must be different in equilibria representing different experiments
! or calculated in parallel or results saved from step or map.
! nextfree: In unused phase_varres record it is the index to next free record
!    The global integer csfree is the index of the first free record
!    The global integer highcs is the higest varres index used
! phlink: is index of phase record for this phase_varres record
! status2: has phase status bits like ENT/FIX/SUS/DORM
! phstate: indicate state: fix/stable/entered/unknown/dormant/suspended/hidden
!                          2   1      0        -1      -2      -3       -4
! phtupx: phase tuple index
     integer nextfree,phlink,status2,phstate,phtupx
! abnorm(1): amount moles of atoms for a formula unit of the composition set
! abnorm(2): mass/formula unit (both set by call to set_constitution)
! prefix and suffix are added to the name for composition sets 2 and higher
     double precision, dimension(2) :: abnorm
     character*4 prefix,suffix
```

```
! constat: array with status word for each constituent, any can be suspended
! yfr: the site fraction array
! mmyfr: min/max fractions, negative is a minumum
! sites: site ratios (which can vary for ionic liquids)
     integer, dimension(:), allocatable :: constat
     double precision, dimension(:), allocatable :: yfr
     real, dimension(:), allocatable :: mmyfr
     double precision, dimension(:), allocatable :: sites
! for ionic liquid derivatives of sites wrt fractions (it is the charge),
! 2nd derivates only when one constituent is vacancy
! 1st sublattice P=\sum_j (-v_j)*y_j + Qy_Va
! 2nd sublattice Q=\sum_i v_i*y_i
! dpqdy is the abs(valency) of the species, set in set_constitution
! for the vacancy it is the same as the number of sites on second subl.
! used in the minimizer and maybe elsewhere
     double precision, dimension(:), allocatable :: dpqdy
     double precision, dimension(:), allocatable :: d2pqdvay
! disfra: a structure describing the disordered fraction set (if any)
! for extra fraction sets, better to go via phase record index above
! this TYPE(gtp_fraction_set) variable is a bit messy.  Declaring it in this
! way means the record is stored inside this record.
     type(gtp_fraction_set) :: disfra
! ---
! arrays for storing calculated results for each phase (composition set)
! amfu: is amount formula units of the composition set (calculated result)
! netcharge: is net charge of phase
! dgm: driving force
     double precision amfu,netcharge,dgm
! Other properties may be that: gval(*,2) is TC, (*,3) is BMAG, see listprop
! nprop: the number of different properties (set in allocate)
!- ncc: total number of site fractions (redundant but used in some subroutines)
! BEWHARE: ncc seems to be wrong using TQ test program fenitq.F90 ???
! listprop(1): is number of calculated properties
! listprop(2:listprop(1)): identifies the property stored in gval(1,ipy) etc
!   2=TC, 3=BMAG. Properties defined in the gtp_propid record
!-     integer nprop,ncc
     integer nprop
     integer, dimension(:), allocatable :: listprop
! gval etc are for all composition dependent properties, gval(*,1) for G
! gval(*,1): is G, G.T, G.P, G.T.T, G.T.P and G.P.P
! dgval(1,j,1): is first derivatives of G wrt fractions j
! dgval(2,j,1): is second derivatives of G wrt fractions j and T
! dgval(3,j,1): is second derivatives of G wrt fractions j and P
! d2gval(ixsym(i,j),1): is second derivatives of G wrt fractions i and j
     double precision, dimension(:,:), allocatable :: gval
     double precision, dimension(:,:,:), allocatable :: dgval
```

```
     double precision, dimension(:,:), allocatable :: d2gval
! added for strain/stress, current values of lattice parameters
     double precision, dimension(3,3) :: curlat
! saved values from last equilibrium for dot derivative calculations
     double precision, dimension(:,:), allocatable :: cinvy
     double precision, dimension(:), allocatable :: cxmol
     double precision, dimension(:,:), allocatable :: cdxmol
  END TYPE gtp_phase_varres
! this record is created inside the gtp_equilibrium_data record
```

6.2.5 The equilibrium record

The equilibrium record has all data that may change dynamically during a calculation. One may have several equilibrium records and during parallel computing each thread must have one. Also during assessments each experimental data is stored in a separate equilibrium record as it has its unique set of conditions.

```
! this must be incremented when a change is made in gtp_equilibrium_data
  INTEGER, parameter :: gtp_equilibrium_data_version=3
  TYPE gtp_equilibrium_data
! this contains all data specific to an equilibrium like conditions,
! status, constitution and calculated values of all phases etc
! Several equilibria may be calculated simultaneously in parallel threads
! so each equilibrium must be independent
! NOTE: the error code must be local to each equilibria!!!!
! During step and map each equilibrium record with results is saved
! values of T and P, conditions etc.
! Values here are normally set by external conditions or calculated from model
! local list of components, phase_varres with amounts and constitution
! lists of element, species, phases and thermodynamic parameters are global
! status: not used yet?
! multiuse: used for various things like direction in start equilibria
! eqno: sequential number assigned when created
! next: index of next equilibrium in a sequence during step/map calculation.
! eqname: name of equilibrium
! comment: a free text, for example reference for experimental data.
! tpval(1) is T, tpval(2) is P, rgas is R, rtn is R*T
! rtn: value of R*T
! weight: weight value for this experiment, default unity
     integer status,multiuse,eqno,next
     character eqname*24,comment*72
     double precision tpval(2),rtn
     double precision :: weight=one
! svfunres: the values of state variable functions valid for this equilibrium
     double precision, dimension(:), allocatable :: svfunres
```

```
! the experiments are used in assessments and stored like conditions
! lastcondition: link to condition list
! lastexperiment: link to experiment list
     TYPE(gtp_condition), pointer :: lastcondition,lastexperiment
! components and conversion matrix from components to elements
! complist: array with components
! compstoi: stoichiometric matrix of compoents relative to elements
! invcompstoi: inverted stoichiometric matrix
     TYPE(gtp_components), dimension(:), allocatable :: complist
     double precision, dimension(:,:), allocatable :: compstoi
     double precision, dimension(:,:), allocatable :: invcompstoi
! one record for each phase+composition set that can be calculated
! phase_varres: here all calculated data for the phase is stored
     TYPE(gtp_phase_varres), dimension(:), allocatable :: phase_varres
! index to the tpfun_parres array is the same as in the global array tpres
! eq_tpres: here local calculated values of TP functions are stored
     TYPE(tpfun_parres), dimension(:), pointer :: eq_tpres
! current values of chemical potentials stored in component record but
! duplicated here for easy acces by application software
     double precision, dimension(:), allocatable :: cmuval
! xconc: convergence criteria for constituent fractions and other things
     double precision xconv
! delta-G value for merging gridpoints in grid minimizer
! smaller value creates problem for test step3.BMM, MC and austenite merged
     double precision :: gmindif=-5.0D-2
! maxiter: maximum number of iterations allowed
     integer maxiter
! This is to store additional things not really invented yet ...
! It may be used in ENTER MANY_EQUIL for things to calculate and list
     character (len=80), dimension(:), allocatable :: eqextra
! this is to save a copy of the last calculated system matrix, needed ??
! to calculate dot derivatives, initiate to zero
     integer :: sysmatdim=0,nfixmu=0,nfixph=0
     integer, allocatable :: fixmu(:)
     integer, allocatable :: fixph(:,:)
     double precision, allocatable :: savesysmat(:,:)
  END TYPE gtp_equilibrium_data
! The primary copy of this structures is declared globally as FIRSTEQ here
! Others may be created when needed for storing experimental data or
! for parallel processing. A global array of these are
  TYPE(gtp_equilibrium_data), dimension(:), allocatable, target :: eqlista
  TYPE(gtp_equilibrium_data), pointer :: firsteq
! This array of equilibrium records are used for storing results during
! STEP and MAP calculations.
  TYPE(gtp_equilibrium_data), dimension(:), allocatable :: eqlines
```

6.3 Records with data shared by several subroutines

The records below have no global variables but are used in some of the calculating subroutines to store complex temporary data, almost like an old-fashioned COMMON area but it is declared inside a subroutine and passed as an argument to the different subroutines.

6.3.1 Parsing data

The data in this record is used parsing the endmember lista and the binary interaction tree.

```
! for each permutation in the binary interaction tree of an endmember one must
! keep track of the permutation and the permutation limit.
! It is not possible to push the value on pystack as one must remember
! them when changing the endmember permutation
! integer, parameter :: permstacklimit=150
! this constant must be incremented when a change is made in gtp_parcalc
  INTEGER, parameter :: gtp_parcalc_version=1
  TYPE gtp_parcalc
! This record contains temporary data that must be separate in different
! parallel processes when calculating G and derivatives for any phase.
! There is nothing here that need to be saved after the calculation is finished
! global variables used when calculating G and derivaties
! sublattice with interaction, interacting constituent, endmember constituents
! PRIVATE inside this structure not liked by some compilers....
! endcon must have maxsubl dimension as it is used for all phases
     integer :: intlat(maxinter),intcon(maxinter),endcon(maxsubl)
! interaction level and number of fraction variables
     integer :: intlevel,nofc
! interacting constituents (max 4) for composition dependent interaction
! iq(j) indicate interacting constituents
! for binary RK+Muggianu iq(3)=iq(4)=iq(5)=0
! for ternary Muggianu in same sublattice iq(4)=iq(5)=0
! for reciprocal composition dependent iq(5)=0
! for Toop, Kohler and simular iq(5) non-zero (not implemented)
     integer :: iq(5)
! fraction variables in endmember (why +2?) and interaction
     double precision :: yfrem(maxsubl+2),yfrint(maxinter)
! local copy of T, P and RT for this equilibrium
     double precision :: tpv(2),rgast
!    double precision :: ymin=1.0D-30
  end TYPE gtp_parcalc
! this record is declared locally in subroutine calcg_nocheck
```

6.3.2 Fraction product stack

The product of the constituent fractions and their derivatives must be saved now and again during the parsing. The record below is used for that.

```
! this constant must be incremented when a change is made in gtp_pystack
  INTEGER, parameter :: gtp_pystack_version=1
  TYPE gtp_pystack
! records created inside the subroutine push/pop_pystack
! data stored during calculations when entering an interaction record
! previous: link to previous record in stack
! ipermutsave: permutation must be saved
! intrecsave: link to interaction record
! pysave: saved value of product of all constituent fractions
! dpysave: saved value of product of all derivatives of constituent fractions
! d2pysave: saved value of product of all 2nd derivatives of constit fractions
     TYPE(gtp_pystack), pointer :: previous
     integer :: pmqsave
     TYPE(gtp_interaction), pointer :: intrecsave
     double precision :: pysave
     double precision, dimension(:), allocatable :: dpysave
     double precision, dimension(:), allocatable :: d2pysave
  end TYPE gtp_pystack
! declared inside the calcg_internal subroutine
```

6.4 STEP and MAP results data structures

Some application software which is closely related to the equilibrium calculation, like STEP and MAP have some of their data structures declared here as they would otherwise not be able to access the data.

6.4.1 The node point record

In the gtp_eqnode record an equilibrium representing a node point with several lines meeting is stored. It has links to other node points and to two or more lines of calculated equilibria.

```
  INTEGER, parameter :: gtp_eqnode_version=1
  TYPE gtp_eqnode
! This record is to arrange calculated equilibria, for example results
! from a STEP or MAP calculation, in an ordered way.  The equilibrium records
! linked from an eqnode record should normally represent one or more lines
! in a diagram but may be used for other purposes.
! ident is to be able to find a specific node
! nodedtype is to specify invariant, middle, end etc.
```

```
! status can be used to supress a line
! color can be used to sepecify color or linetypes (dotted, thick ... etc)
! exits are the number of lines that should exit from the node
! done are the number of calculated lines currently exiting from the node
     integer ident,nodetype,status,color,exits,done
! this node can be in a multilayerd list of eqnodes
     type(gtp_eqnode), pointer :: top,up,down,next,prev
! nodeq is a pointer to the equilibrium record at the node
     type(gtp_equilibrium_data), pointer :: nodeq
! eqlista are pointers to line of equilibria starting or ending at the node
! The equilibrium records are linked with a pointer inside themselves
     type(gtp_equilibrium_data), dimension(:), pointer :: eqlista
! axis is the independent axis variable for the line, negative means decrement
! noeqs gives the number of equilibria in each eqlista, a negative value
! indicates that the node is an endpoint (each line normally has a
! start point and an end point)
     integer, dimension(:), allocatable :: axis,noeqs
! This is a possibility to specify a status for each equilibria in each line
!    integer, dimension(:,:), allocatable :: eqstatus
  end TYPE gtp_eqnode
! can be allocated in a gtp_applicationhead record
```

6.5 Assessments

Multicomponent thermodynamic databases are created by assessments of many binary and ternary systems starting from the pure elements. In an assessment experimental and theoretical data are fitted by model parameters for the individual phases. In order to create large databases and extrapolate to multicomponent systems the data and models used for the lower order systems, in particular the pure elements, must be identical.

The assessment of a phase requires that one can include experimental data in the data structure and by using an optimizing software that can vary some of the model parameters. The necessary data structure for this is provided in the GTP package.

The setup of an assessment will include the commands (note the commands listed below may change in later versions of the user interface):

1. ENTER elements, species, phases etc. usually from a macro file.
2. ENTER OPT to enter the coefficients to be optimized. Part of the TPfun package.
3. ENTER PARAMETERS with coefficients to be optimized.
4. ENTER EQUILIBRIA with experimental data (possibly from a file). The experimental data for each equilibrium added with ENTER EXPERIMENT command, see section 8.9.5
5. SET RANGE_EXP_EQUIL to specify the equilibria with experiments.

6. SET VARIABLE_COEFF to specify coefficients to be optimized.
7. OPTIMIZE to make a least square fit.
8. LIST OPTIMIZATION to list the result
9. SAVE TDB to create a TDB file with the results (need editing).
10. and other commands as necessry

6.5.1 The assessment head record

This is the record organizing an assessment. It has the all the values related to the optimizing coefficients and has a list of pointers to the equilibria with experimental data. An instance of this is created when starting the software and thus accessible in all packages that use GTP, the arrays declared inside the record can be updated by such packages. There is a possibility to create a linked list of these records to save temporary versions of an assessment. This is not yet implemented and some of the other variables in this recored are not yet used.

The status word has just a single bit defined, AHCOEF, set if the optimizing coefficients have been entered by the command ENTER OPTIMIZE_COEFF. The assessment coefficients are TP function constants named A00 to A99. These TP constants can be used entering other TP functions and phase parameters that should be optimized. There are also subroutines to change the values of these variables from an optimizing software. In order to control the optimization the user can also provide a scaling factor, a minimum and a maximum of a coefficient.

The eqlista array contain pointers to the equilibria with experimental data to be used in the assessment. This is set by a command in the user interface, SET RANGE_EXP_EQUIL. The equilibria with experiments are simply entered with an ENTER EQUILIBRIUM command and for each equilibrium the associated experimental data is entered with the command ENTER EXPERIMENT.

```
! a smart way to have an array of pointers used in gtp_assessmenthead
  TYPE equilibrium_array
     type(gtp_equilibrium_data), pointer :: p1
  end TYPE equilibrium_array
  INTEGER, parameter :: gtp_assessment_version=1
  TYPE gtp_assessmenthead
! This record should summarize the essential information about assessment data
! using GTP.  How it should link to other information is not clear.
! status is status word, AHCOEF is used
! varcoef is the number of variable coefficients
! firstexpeq is the first equilibrium with experimental data
     integer status,varcoef,firstexpeq
     character*64 general,special
     type(gtp_assessmenthead), pointer :: nextash,prevash
! This is list of pointers to equilibria to be used in the assessnent
```

```
! size(eqlista) is the number of equilibria with experimental data
     type(equilibrium_array), dimension(:), allocatable :: eqlista
! These are the coefficients values that are optimized,
! current values, scaling, start values and optionally min and max
     double precision, dimension(:), allocatable :: coeffvalues
     double precision, dimension(:), allocatable :: coeffscale
     double precision, dimension(:), allocatable :: coeffstart
     double precision, dimension(:), allocatable :: coeffmin
     double precision, dimension(:), allocatable :: coeffmax
! These are the corresponding TP-function constants indices
     integer, dimension(:), allocatable :: coeffindex
! This array indicate currently optimized variables:
!  0=fix, 1=fix with min, 2=fix with max, 3=fix with min and max
!  10=optimized, 11=opt with min, 12=opt with max, 13=opt with min and max
     integer, dimension(:), allocatable :: coeffstate
! Work arrays ...
     double precision, dimension(:), allocatable :: wopt
  end TYPE gtp_assessmenthead
! this record is allocated when necessary
  type(gtp_assessmenthead), pointer :: firstash,lastash
```

6.6 Other application head record

This is a template for other applications that need access to the internal data of OC.

```
  INTEGER, parameter :: gtp_applicationhead_version=1
  TYPE gtp_applicationhead
! This record should summarize the essential information about an application
! using GTP.  How it should link to other information is not clear.
! The character variables should be used to indicate that.
     integer apptyp,status
     character*64 general,special
! These can be used to define axis and other things
     integer, dimension(:), allocatable :: ivals
     double precision, dimension(:), allocatable :: rvals
     character*64, dimension(:), allocatable :: cvals
     type(gtp_applicationhead), pointer :: nextapp,prevapp
! The headnode can be the start of a structure of eqnodes with lines
     type(gtp_eqnode) :: headnode
! this is the start of a list of nodes with calculated lines or
! single equilibria that belong to the application.
     type(gtp_eqnode), dimension(:), allocatable :: nodlista
  end TYPE gtp_applicationhead
! this record is allocated when necessary
  type(gtp_applicationhead), pointer :: firstapp,lastapp
```

7 Global variables

The variables below contain some information used in several subroutines. Additionally many of the record types described earlier are declared as private arrays to protect them somewhat.

```
! counters for elements, species and phases initiated to zero
  integer, private :: noofel=0,noofsp=0,noofph=0
! counter for phase tuples (combination of phase+compset)
  integer, private :: nooftuples=0
! counters for property and interaction records, just for fun
  integer, private :: noofprop,noofint,noofem
! free lists in phase_varres records and addition records
  integer, private :: csfree,addrecs
! free list of references and equilibria
  integer, private :: reffree,eqfree
! maximum number of properties calculated for a phase
  integer, private :: maxcalcprop=20
! highcs is highest used phase_varres record (for copy equil etc)
  integer, private :: highcs
! Trace for debugging (not used)
  logical, private :: ttrace
! minimum constituent fraction
  double precision :: bmpymin
! number of defined property types like TC, BMAG etc
  integer, private :: ndefprop
```

8 Subroutines and functions

It is not self evident how to organize the description of the subroutines in GTP. One can base it on the type of service the subroutine provides like entering data, listing data, calculating or on the type of object the action is performed on like on elements, phases, additions etc. A mixed approach is made here where '"find", 'enter" and "list" subroutines operating on general objects are grouped together whereas subroutines specific for "state variables", "additions" and similar things are grouped together for all services like entering, listing and calculations. One reason is that when a new addition is implemented all of these services must be provided together so it is natural to keep them together also in the documentation as a programmer has to provide new subroutines for all of them.

8.1 Variable names

The arguments for the subroutines and functions are normally explained but some standards have been used

Symbol	Type	Meaning
iph	integer	Phase index in PHASES
ics	integer	Composition set number
lokph	integer	Phase location (index to PHLISTA)
lokcs	integer	Index of PHASE_VARRES array for a composition set
ceq	pointer	Pointer to current gtp_equilibrium_data record

8.2 Initiallization

This subroutine must be called before any other in the GTP package. It dimensions arrays and creates some initial data structures. The arguments are presently not used but should be used to dimension arrays and provide default values.

```
 subroutine init_gtp(intvar,dblvar)
! initiate the data structure
! create element and species record for electrons and vacancies
! the allocation of many arrays should be provided calling this routne
! intvar and dblvar will eventually be used for allocations and defaults
   implicit none
   integer intvar(*)
   double precision dblvar(*)
```

8.2.1 Initialize the assessment

This is maybe redundant.

```
 subroutine assessmenthead(ash)
! create an assessment head record
   type(gtp_assessmenthead), pointer :: ash
```

8.3 Functions to know how many

The counters for elements etc are private so external software must call functions to find out how many elements etc that the system has. They are all given here.

```
 integer function noel()
! number of elements because noofel is private
! should take care if elements are suspended
 integer function nosp()
! number of species because noofsp is private
! should take care if species are suspended
 integer function noph()
! number of phases because noofph is private
```

```
! should take care if phases are hidden
 integer function noofcs(iph)
! returns the number of compositions sets for phase iph
   implicit none
   integer iph
 integer function noconst(iph,ics,ceq)
! number of constituents for iph (include single constituents on a sublattice)
! It tests if a constituent is suspended which can be different in each ics.
   implicit none
   integer iph,ics
   TYPE(gtp_equilibrium_data), pointer :: ceq
 integer function nooftup()
! number of phase tuples
```

8.3.1 How many state variable functions

The number of state variable functions entered is given by this.

```
 integer function nosvf()
! number of state variable functions
```

8.3.2 How many equilibria

There is a global array with equilibria records but only a few of them may be allocated with data. This routine returns the number of allocated equilibrium records.

```
 integer function noeq()
! returns the number of equilibria entered
```

8.3.3 Total number of phases and composition sets

This is needed when dimensioning arrays for phases and composition sets for calculations.

```
 integer function nonsusphcs(ceq)
! returns the total number of unhidden phases+composition sets
! in the system.  Used for dimensioning work arrays and in loops
   implicit none
   TYPE(gtp_equilibrium_data), pointer :: ceq
```

8.4 Find things

These subroutines translate from name to index or location of data or vice versa. There are other some "find" subroutines for special things like find_gridmin described in 8.17.6.

```
 subroutine find_element_by_name(name,iel)
! find an element index by its name, exact fit required
   implicit none
   character name*(*)
   integer iel
 subroutine find_component_by_name(name,icomp,ceq)
! BEWARE: one may in the future have different components in different
! equilibria. components are a subset of the species
   implicit none
   character*(*) name
   integer icomp
   TYPE(gtp_equilibrium_data), pointer :: ceq
 subroutine find_species_by_name(name,isp)
! locates a species index from its name, unique abbreviation
! or exact match needed
   implicit none
   character name*(*)
   integer isp
 subroutine find_species_record(name,loksp)
! locates a species record allowing abbreviations
   implicit none
   character name*(*)
   integer loksp
 subroutine find_species_record_noabbr(name,loksp)
! locates a species record no abbreviations allowed
   implicit none
   character name*(*)
   integer loksp
 subroutine find_species_record_exact(name,loksp)
! locates a species record, exact match needed
! for parameters, V must not be accepted as abbreviation of VA or C for CR
   implicit none
   integer loksp
   character name*(*)
 subroutine find_phasetuple_by_name(name,phcsx)
! finds a phase with name "name", returns phase tuple index
! handles composition sets either with prefix/suffix or #digit
! When no pre/suffix nor # always return first composition set
   implicit none
   character name*(*)
   integer phcsx
 subroutine find_phase_by_name(name,iph,ics)
! finds a phase with name "name", returns address of phase, first fit accepted
! handles composition sets either with prefix/suffix or #digit
! When no pre/suffix nor # always return first composition set
   implicit none
```

```
   character name*(*)
   integer iph,ics
 subroutine find_phasex_by_name(name,phcsx,iph,ics)
! finds a phase with name "name", returns index and tuplet of phase.
! All phases checked and error return if name is ambiguous
! handles composition sets either with prefix/suffix or #digit or both
! if no # check all composition sets for prefix/suffix
   implicit none
   character name*(*)
   integer phcsx,iph,ics
 subroutine find_phase_by_name_exact(name,iph,ics)
! finds a phase with name "name", returns address of phase. exact match req.
! handles composition sets either with prefix/suffix or #digit
! no pre/suffix nor # gives first composition set
   implicit none
   character name*(*)
   integer iph,ics
```

8.4.1 Find and select equilibrium

As already stated each equilibrium has a separate set of conditions and results. Equilibria are created by calling enter_equilibrium, see section 8.7.12. A "default" equilibrium pointed to by the variable "firsteq" is created when the program is started. The pointer variable "ceq" is used in many subroutine to indicate the "current" equilibrium for which calculations or data are used.

```
 subroutine findeq(name,ieq)
! finds the equilibrium with name "name" and returns its index
! ieq should be the current equilibrium
   implicit none
   character name*(*)
   integer ieq
 subroutine selecteq(ieq,ceq)
! checks if equilibrium ieq exists and if so set it as current
   implicit none
   TYPE(gtp_equilibrium_data), pointer :: ceq
   integer ieq
```

8.5 Get things

The difference between find and get is not very distinct. Normally the "find" routines requires a name or symbol to return an index or location whereas the "get" routines require an index or location to get more data. Some of the "get" routines are very specific and described together with the type of data you want to get.

```
 subroutine get_phase_record(iph,lokph)
! given phase index iph this returns the phase location lokph
   implicit none
   integer iph,lokph
 subroutine get_phase_variance(iph,nv)
! returns the number of independent variable fractions in phase iph
   implicit none
   integer iph,nv
 subroutine get_constituent_location(lokph,cno,loksp)
! returns the location of the species record of a constituent
! requred for ionic liquids as phlista is private
   implicit none
   integer lokph,loksp,cno
 subroutine get_phase_compset(iph,ics,lokph,lokcs)
! Given iph and ics the phase and composition set locations are returned
! Checks that ics and ics are not outside bounds.
   implicit none
   integer iph,ics,lokph,lokcs
 subroutine find_constituent(iph,spname,mass,icon)
! find the constituent "spname" of a phase. spname can have a sublattice #digit
! Return the index of the constituent in icon.  Additionally the mass
! of the species is returned.
   implicit none
   character*(*) spname
   double precision mass
   integer iph,icon
```

8.5.1 Get phase constituent name

By supplying a phase index and and the sequential index (counted over all sublattices, this routine returns the name and mass of the constituent.

```
 subroutine get_constituent_name(iph,iseq,spname,mass)
! find the constituent with sequential index iseq in phase iph
! return name in "spname" and mass in mass
   implicit none
   character*(*) spname
   integer iph,iseq
   double precision mass
```

8.5.2 Get element data

The data for an element is returned.

```
 subroutine get_element_data(iel,elsym,elname,refstat,mass,h298,s298)
```

```
! return element data as that is stored as private in GTP
   implicit none
   character elsym*2, elname*(*),refstat*(*)
   double precision mass,h298,s298
   integer iel
```

8.5.3 Get component or species name

Components are by default the elements but the user can (sometimes in the future) change this to any set of species (that are orthogonal). Each equilibrium will be able to have a different set of components.

The set of components are important because one can only use components to set amounts of fractions with conditions like $N(<\text{components}>)$. There is an alternative method to set amounts using the subroutine set_input_amounts where amounts or mass of different species can be used to give amounts of components.

```
 subroutine get_component_name(icomp,name,ceq)
! return the name of component icomp
   implicit none
   character*(*) name
   integer icomp
   TYPE(gtp_equilibrium_data), pointer :: ceq
 subroutine get_species_name(isp,spsym)
! return species name, isp is species number
   implicit none
   character spsym*(*)
   integer isp
```

8.5.4 Get species data

A species is just a stoichiometric arrangement of elements like a molecule. It has no thermodynamic data as they are stored together with the phase. The stoichiometry of a species is fixed. The composition of a phase can vary if there are two ore more species as constituents in one or more sublattices.

```
 subroutine get_species_data(loksp,nspel,ielno,stoi,smass,qsp)
! return species data, loksp is from a call to find_species_record
! nspel: integer, number of elements in species
! ielno: integer array, element indices
! stoi: double array, stoichiometric factors
! smass: double, mass of species
! qsp: double, charge of the species
   implicit none
   integer, dimension(*) :: ielno
```

```
   double precision, dimension(*) :: stoi(*)
   integer loksp,nspel
   double precision smass,qsp
```

8.5.5 Mass of component

For the mass balance calculations during equilibrium calculations the mass of a component is needed frequently. This subroutine just returns the mass of a component.

```
 double precision function mass_of(component,ceq)
! return mass of component
! smass: double, mass of species
   implicit none
   integer :: component
   TYPE(gtp_equilibrium_data), pointer :: ceq
```

8.5.6 Get phase name

The title says all but there are two variants of this, one using phase tuples, the other integer variables as arguments. There are also some redundant routines to get the record for phase data.

```
 subroutine get_phase_name(iph,ics,name)
! Given the phase index and composition set number this subroutine returns
! the name with pre- and suffix for composition sets added and also
! a \# followed by a digit 2-9 for composition sets higher than 1.
   implicit none
   character name*(*)
   integer iph,ics
 subroutine get_phasetup_name(phtupx,name)
! phasetuple(phtupx)%phase is index to phlista
! the name has pre- and suffix for composition sets added and also
! a \# followed by a digit 2-9 for composition sets higher than 1.
   implicit none
   character name*(*)
   integer phtupx
 subroutine get_phasetup_name_old(phtuple,name)
! Given the phase tuple this subroutine returns the name with pre- and suffix
! for composition sets added and also a \# followed by a digit 2-9 for
! composition sets higher than 1.
   implicit none
   character name*(*)
   type(gtp_phasetuple) :: phtuple
 subroutine get_phasetup_record(phtx,lokcs,ceq)
```

```
! return lokcs when phase tuple known
   implicit none
   integer phtx,lokcs
   TYPE(gtp_equilibrium_data), pointer :: ceq
```

8.5.7 Get phase data

This is a very important subroutine used at each iteration during calculations to obtain information about a phase. In the knr array the constituents are given as the integer indices of the species location in the array SPLISTA. The constituents are stored sequentially and the first nkl(1) positions in knr belong to sublattice 1, the next nkl(2) to sublattice 2 etc. The order is alphabetical for each sublattice (not for ionic liquid). In yarr the fractions of the constituents are given sequentially in the same order.

NOTE the ionic liquid has the constituents in the second sublattice ordered by first all anions, then Va (if any), then all neutrals. The anions and the neutrals are ordered alphabetically.

In the array qq the current number of components per formula unit is returned in the first index and the current charge (valence) in the second.

```
 subroutine get_phase_data(iph,ics,nsl,nkl,knr,yarr,sites,qq,ceq)
! return the structure of phase iph and constituntion of comp.set ics
! nsl: integer, number of sublattices
! nkl: integer array, number of constituents in each sublattice
! knr: integer array, species location (not index) of constituents (all subl)
! yarr: double array, fraction of constituents (in all sublattices)
! sites: double array, number of sites in each sublattice
! qq: double array, (must be dimensioned at least 5) although only 2 used:
! qq(1) is number of real atoms per formula unit for current constitution
! qq(2) is net charge of phase for current constitution
! ceq: pointer, to current gtp_equilibrium_data record
   implicit none
   integer, dimension(*) :: nkl,knr
   double precision, dimension(*) :: yarr,sites,qq
   integer iph,ics,nsl
   TYPE(gtp_equilibrium_data), pointer :: ceq
```

8.5.8 Phase tuple array

This subroutine is redundant as application software can use the PHASETUPLE array declared within the OC software.

```
 integer function get_phtuplearray(phcs)
! copies the internal phase tuple array to external software
```

```
! function value set to number of tuples
   type(gtp_phasetuple), dimension(*) :: phcs
```

8.6 Set things

Many things can be set but most of the ways to set them are described together with the object to set. How to set conditions is described in 8.9.6.

8.6.1 Set constitution

This is a subroutine used frequently when iterating to find the equilibrium. At each iteration the new constitutions of the phases are set using this subroutine. Some internal quantities are also calculated like the number of atoms per mole formula unit of the phase and the charge of the phase. These are returned in the call.

The array yfr in the phase_varres record belonging to the composition set (iph,ics) is not private and a programmer may thus change the values externally without calling set_constitution. But this is strongly discouraged as the internal variables qq(1) and qq(2) must be updated for each set of fractions to ensure that the massbalance is correct.

The order of the fractions in yfra must be the same as in get_phase_data, see 8.5.7.

```
 subroutine set_constitution(iph,ics,yfra,qq,ceq)
! set the constituent fractions of a phase and composition set and the
! number of real moles and mass per formula unit of phase
! returns number of real atoms in qq(1), charge in qq(2) and mass in qq(3)
! for ionic liquids sets the number of sites in the sublattices
   implicit none
   double precision, dimension(*) :: yfra,qq
   integer iph,ics
   TYPE(gtp_equilibrium_data), pointer :: ceq
```

8.6.2 Set reference state for a component

For each component the user can select a phase, temperature and pressure as reference state. If a * is given as temperature and pressure the current value of T and P will be used. The reference state is used in calculate_reference_state in section 8.11.11.

```
 subroutine set_reference_state(icomp,iph,tpval,ceq)
! set the reference state of a component to be "iph" at tpval
   implicit none
   integer icomp,iph
   double precision, dimension(2) :: tpval
   TYPE(gtp_equilibrium_data), pointer :: ceq
```

8.6.3 Set condition

See section 8.9.6.

8.7 Enter data

The subroutines for entering data and other things can be named as new, enter, add, create etc. according to the mind of the programmer when the subroutine was written. In all cases the data is provided as arguments in the call, there is no interactions with the user.

8.7.1 Enter element data

All data for an element. Some checks are made. The elements are automatically eneterd also as species so they can be constituents of phases.

```
 subroutine enter_element(symb,name,refstate,mass,h298,s298)
! Creates an element record after checks.
! symb: character*2, symbol (it can be a single character like H or V)
! name: character, free text name of the element
! refstate: character, free text name of reference state.
! mass: double, mass of element in g/mol
! h298: double, enthalpy difference between 0 and 298.14 K
! s298: double, entropy at 298.15 K
   implicit none
   CHARACTER*(*) symb,name,refstate
   DOUBLE PRECISION mass,h298,s298
```

8.7.2 Enter species data

All data for an element. Some checks are made. The elements constituting the species must have been entered before. A species can have a positive or negative charge using the element index -1 with a stoichiometric factor.

```
 subroutine enter_species(symb,noelx,ellist,stoik)
! creates a new species
! symb: character*24, name of species, often equal to stoichiometric formula
! noelx: integer, number of elements in stoichiometric formula (incl charge)
! ellist: character array, element names (electron is /-)
! stoik: double array, must be positive except for electron.
   implicit none
   character symb*(*),ellist(*)*(*)
   integer noelx
   double precision stoik(*)
```

8.7.3 Enter phase and model

This subroutine is called with the model data needed to create the data structure for a phase (no parameter data). The model variable is just a text, phtype is used to arrange gas (G) and liquids (L) before the alphabetical list of the other phases.

```
 subroutine enter_phase(name,nsl,knr,const,sites,model,phtype)
! creates the data structure for a new phase
! name: character*24, name of phase
! nsl: integer, number of sublattices (range 1-9)
! knr: integer array, number of constituents in each sublattice
! const: character array, constituent (species) names in sequential order
! sites: double array, number of sites on the sublattices
! model: character, free text
! phtype: character*1, specifies G for gas, L for liquid
   implicit none
   character name*(*),model*(*),phtype*(*)
   integer nsl
   integer, dimension(*) :: knr
   double precision, dimension(*) :: sites
   character, dimension(*) :: const*(*)
```

8.7.4 Sorting constituents in ionic liquids

The ionic liquid model requires all cations (with positive charge) on the first sublattice in alphabetical order. On the second sublattice the anions (with negative charge) should be first (in alphabetical order), the the hypothetical vacancy (if any), then any neutrals in alphabetical order. This subroutine takes care of that

```
 subroutine sort_ionliqconst(lokph,mode,knr,kconlok,klok)
! sorts constituents in ionic liquid, both when entering phase
! and decoding parameter constituents
! order: 1st sublattice only cations
! 2nd: anions, VA, neutrals
! mode=0 at enter phase, wildcard ok in 1st sublattice if neiher anions nor Va
! mode=1 at enter parameter (wildcard allowed, i.e. some kconlok(i)=-1)
! some  parameters not allowed, L(ion,A+:B,C), must be L(ion,*:B,C), check!
   implicit none
   integer lokph,knr(*),kconlok(*),klok(*),mode
```

8.7.5 Enter composition set

As explained in section 2.6 a phase may exist simultaneously with several different composition sets. This can be due to miscibility gaps or ordering. Some carbides like cubic TiC is

modeled as the same phase as the metallic FCC and it may be stable at the same time as the austenite phase in steels. This subroutine creates a new composition set for a phase.

```
 subroutine enter_composition_set(iph,prefix,suffix,icsno)
! adds a composition set to a phase.
! iph: integer, phase index
! prefix: character*4, optional prefix to original phase name
! suffix: character*4, optional suffix to original phase name
! icsno: integer, returned composition set index (value 2-9)
! ceq: pointer, to current gtp_equilibrium_data
!
! BEWARE this must be done in all equilibria (also during parallel processes)
! There may still be problems with equilibria saved during STEP and MAP
!
   implicit none
   integer iph,icsno
   character*(*) prefix,suffix
!   TYPE(gtp_equilibrium_data), pointer :: ceq
```

8.7.6 Remove or suspend composition set

Sometimes the grid minimizer creates too many composition sets and the further calculations may be easier if these are removed. But sometimes it is not possible to delete or remove them (when running in parallel) and then they may be suspended.

```
 subroutine suspend_composition_set(iph,parallel,ceq)
! the last composition set is suspended in all equilibria
!
! If parallel is TRUE then execution is not in parallel (threaded)
!
   implicit none
   integer iph
   logical parallel
   type(gtp_equilibrium_data), pointer :: ceq
 subroutine remove_composition_set(iph,force)
! the last composition set is deleted, update csfree and highcs
!
! >>>>>>>>>>>>>>>>>>>>>>>>>>>> NOTE <<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<< !
! Not safe to remove composition sets when more than one equilibrium       !
! >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>> !
!
! If force is TRUE delete anyway ... very dangerous ...
!
   implicit none
!
```

```
! BEWARE must be for all equilibria but maybe not allowed when threaded
!
   integer iph,jl,tuple
   logical force
```

8.7.7 Enter parameter

All kind of parameters are entered by this subroutine. Called when reading a TDB file or entered interactively, see 8.9.2.

```
 subroutine enter_parameter(lokph,typty,fractyp,nsl,endm,nint,lint,ideg,&
      lfun,refx)
! enter a parameter for a phase from database or interactivly
! typty is the type of property, 1=G, 2=TC, ... , n*100+icon MQ&const#subl
! fractyp is fraction type, 1 is site fractions, 2 disordered fractions
! nsl is number of sublattices
! endm has one constituent index for each sublattice
! constituents in endm and lint should be ordered so endm has lowest
! (done by decode_constarr)
! nint is number of interacting constituents (can be zero)
! lint is array of sublattice+constituent indices for interactions
! ideg is degree
! lfun is link to function (integer index)
! refx is reference (text)
! if this is a phase with permutations all interactions should be in
! the first or the first two identical sublattices (except interstitals)
! a value in endm can be negative to indicate wildcard
! for ionic liquid constituents must be sorted specially
   implicit none
   integer, dimension(*) :: endm
   character refx*(*)
   integer lokph,fractyp,typty,nsl,nint,ideg,lfun
   integer, dimension(2,*) :: lint
```

8.7.8 Subroutines handling fcc permutations

These subroutines creates all possible permutations of parameters for a 4 sublattice fcc phase. The 4 ordering sublattices must be the first and they represent the tetrahedron in the lattice. The number of sites must be the same and the constituents also. There can be additional sublattices for interstitials.

```
 subroutine fccpermuts(lokph,nsl,iord,noperm,elinks,nint,jord,intperm,intlinks)
! finds all fcc/hcp permutations needed for this parameter
! The order of elements in the sublattices is irrelevant when one has F or B
```

```
! ordering as all permutations are stored in one place (with some exceptions)
! Thus the endmembers are ordered alphabetically in the sublattices and also
! the interaction parameters.  Max 2 levels of interactions allowed.
   implicit none
   integer, dimension(*) :: iord,intperm
   integer, dimension(2,*) :: jord
   integer lokph,nsl,noperm,nint
 subroutine fccip2A(lokph,jord,intperm,intlinks)
! 2nd level interaction permutations for fcc
   implicit none
   integer, dimension(*) :: intperm
   integer, dimension(2,*) :: jord,intlinks
   integer lokph
 subroutine fccip2B(lq,lokph,lshift,jord,intperm,intlinks)
! 2nd level interaction permutations for fcc
   implicit none
   integer lq,lokph,lshift
   integer, dimension(*) :: intperm
   integer, dimension(2,*) :: jord,intlinks
 subroutine fccint31(jord,lshift,intperm,intlinks)
! 1st level interaction in sublattice l1 with endmember A:A:A:B or A:B:B:B
! set the sublattice and link to constituent for each endmember permutation
! 1st permutation of endmember: AX:A:A:B; A:AX:A:B; A:A:AX;B  4      0 1 2
! 2nd permutation of endmember: AX:A:B:A; A:AX:B:A; A:A:B:AX  3      0 1 3
! 3rd permutation of endmember: AX:B:A:A; A:B:AX:A; A:B:A:AX  3      0 2 3
! 4th permutation of endmember: B:AX:A:A; B:A:AX:A; B:A:A:AX  1 or   1 2 3
! 1st permutation of endmember: A:BX:B:B; A:B:BX:B; A:B:B:BX  4      0 1 2
! 2nd permutation of endmember: BX:A:B:B; B:A:BX:B; B:A:B:BX  1 etc -1 1 2
! 3rd -1 0 2 ; -1 0 1
! suck
   implicit none
   integer lshift
   integer, dimension(2,*) :: jord,intlinks
   integer, dimension(*) :: intperm
 subroutine fccint22(jord,lshift,intperm,intlinks)
! 1st level for endmember A:A:B:B with interaction in sublattice jord(1,1)
! 6 permutations of endmember, 2 permutations of interactions, 12 in total
! 1st endmemperm: AX:A:B:B; A:AX:B:B      0  1
! 2nd endmemperm: AX:B:A:B; A:B:AX:B      0  2
! 3rd endmemperm: AX:B:B:A; A:B:B:AX      0  3
! 4th endmemperm: B:AX:B:A; B:A:B:AX      1  3
! 5th endmemperm: B:B:AX:A; B:B:A:AX      2  3
! 6th endmemperm: B:AX:A:B; B:A:AX:B or   1  2
! 1th endmemperm: A:A:BX:B; A:A:B:BX      0  1
! 2nd endmemperm: A:BX:A:B; A:B:A:BX     -1  1
! 3rd endmemperm: A:BX:B:A; A:B:BX:A     -1  0
```

```
! 4th endmemperm: BX:A:B:A; B:A:BX:A     -2  0
! 5th endmemperm: BX:B:A:A; B:BX:A:A     -2 -1
! 6th endmemperm: BX:A:A:B; B:A:A:BX     -2  1
   implicit none
   integer lshift
   integer, dimension(2,*) :: jord,intlinks
   integer, dimension(*) :: intperm
 subroutine fccint211(a211,jord,lshift,intperm,intlinks)
! 1st level interaction in sublattice l1 with endmember like A:A:B:C
! 12 endmember permutations of AABC; ABBC; or ABCC
! 2 interaction permutations for each, 24 in total
   implicit none
   integer a211,lshift
   integer, dimension(2,*) :: jord,intlinks
   integer, dimension(*) :: intperm
 subroutine fccpe211(l1,elinks,nsl,lshift,iord)
! sets appropriate links to constituents for the 12 perumations of
! A:A:B:C (l1=1), A:B:B:C (l1=2) and A:B:C:C (l1=3)
   implicit none
   integer, dimension(nsl,*) :: elinks
   integer, dimension(*) :: iord
   integer l1,nsl,lshift
 subroutine fccpe1111(elinks,nsl,lshift,iord)
! sets appropriate links to 24 permutations when all 4 constituents different
! A:B:C:D
! The do loop keeps the same constituent in first sublattice 6 times, changing
! the other 3 sublattice, then changes the constituent in the first sublattice
! and goes on changing in the other 3 until all configurations done
   implicit none
   integer, dimension(nsl,*) :: elinks
   integer, dimension(*) :: iord
   integer nsl,lshift
```

8.7.9 Subroutines handling bcc permutations

Not implemented yet.

```
 subroutine bccpermuts(lokph,nsl,iord,noperm,elinks,nint,jord,intperm,intlinks)
! finds all bcc permutations needed for this parameter
   implicit none
   integer lokph,nsl,noperm,nint
   integer, dimension(*) :: iord,intperm
   integer, dimension(2,*) :: jord
   integer, dimension(:,:), allocatable :: elinks
   integer, dimension(:,:), allocatable :: intlinks
```

8.7.10 Find constituent

```
 subroutine findconst(lokph,ll,spix,constix)
! locates the constituent index of species with index spix in sublattice ll
! and returns it in constix.  For wildcards spix is -99; return -99
! THERE MAY ALREADY BE A SIMULAR SUBROUTINE ... CHECK
   implicit none
   integer lokph,ll,spix,constix
```

8.7.11 Enter references for parameter data

```
 subroutine tdbrefs(refid,line,mode,iref)
! store a reference from a TDB file or given interactivly
! If refid already exist and mode=1 then amend the reference text
   implicit none
   character*(*) refid,line
   integer mode,iref
```

8.7.12 Enter equilibrium

The equilibrium record, as explained in section 6.2.5 has all data necessary for specifying an equilibrium: conditions, compoenets, phases etc. One may have several equilibria with different sets of conditions but they have the same set of phases (the phase status and set of stable phases may differ). This can be used to assess many different experiments or used in simulations where each gridpoint is connected to an equilibrium record. This simplifies parallel processing as each thread can work independently on the data in the equilibrium record.

```
 subroutine enter_equilibrium(name,number)
! creates a new equilibrium.  Allocates arrayes for conditions
! components, phase data and results etc.
! returns index to new equilibrium record
! THIS CAN PROBABLY BE SIMPLIFIED, especially phase_varres array can be
! copied as a whole, not each record structure separately ... ???
   implicit none
   character name*(*)
   integer number
```

8.7.13 Enter many equilibria

This command is specially designed for entering table of experimental data where most conditions are the same but only some values are different. It will ask for a table head which must contain all information needed to calculate the equilibrium. By default all phases are suspended so first give the set of phases to be considered (entered, fixed or dormant). The

line with the phase status start with the status, for fix and entered followed by a number and then a list of phases. An asterisk, *, can be used for all phases.

The conditions can be set after just the word "conditions" and experiments after the word "experiment". It is also possible to set reference states and to demand that some symbols are calculated or values of state variables and parameter identifiers listed.

Each line may refer to columns in the table to follow after the head. The columns are specified with teh "@" character followed by the column. Max 9 columns allowed.

The head is finished by the command "table_start" and then on each line the column values must be given. But first on each line there must be a name of the equilibrium (this is not counted as a column, or as column zero).

The lines with values is terminated by a "table_end".

All equilibria in a table can be calculated with the "calculate all" command after the range of equilibria has been set by the "set range" command.

```
 subroutine enter_many_equil(cline,last)
! executes an enter many_equilibria command
! and creates many similar equilibria from a table
   implicit none
   character*(*) cline
   integer last
```

8.7.14 Delete equilibrium

This is needed after STEP or MAP to clean up the structure as all equilibria along the lines are saved as equilibrium records.

```
 subroutine delete_equilibrium(name,ceq)
! deletes an equilibrium (needed when repeated step/map)
! name can be an abbreviation line "_MAP*"
! deallocates all data.  Minimal checks ... one cannot delete "ceq"
   implicit none
   character name*(*)
   type(gtp_equilibrium_data), pointer :: ceq
```

8.7.15 Copy equilibrium

As part of STEP and MAP equilibrium records are copied between different lists.

```
 subroutine copy_equilibrium(neweq,name,ceq)
! creates a new equilibrium which is a copy of ceq.
! Allocates arrayes for conditions,
! components, phase data and results etc. from equilibrium ceq
```

```
! returns a pointer to the new equilibrium record
! THIS CAN PROBABLY BE SIMPLIFIED, especially phase_varres array can be
! copied as a whole, not each record structure separately ... ???
   implicit none
   character name*(*)
   integer number
   type(gtp_equilibrium_data), pointer ::neweq,ceq
```

8.7.16 Copy condition

This is also a utility used in MAP and STEP

```
 subroutine copy_condition(newrec,oldrec)
! Creates a copy of the condition record "oldrec" and returns a link
! to the copy in newrec.  The links to "next/previous" are nullified
   implicit none
   type(gtp_condition), pointer :: oldrec
   type(gtp_condition), pointer :: newrec
```

8.7.17 Check that a phase is allowed to have fcc permutations

Some minimal checks made.

```
 logical function check_minimal_ford(lokph)
! some tests if the fcc/bcc permutation model can be applied to this phase
! The function returns FALSE if the user may set the FORD or BORD bit of lokph
   implicit none
   integer lokph
```

8.7.18 Calculate new highcs value when a composition set is created or deleted

Highcs is used when saving the allocated phase_varres records and in some other cases. Note that there can be unused phase_varres records below highcs.

```
 integer function newhighcs(reserved)
! updates highcd and arranges csfree to be in sequential order
! highcs is the higest used varres record before the last reservation
! or release of a record.  release is TRUE if a record has been released
! csfree is the beginning of the free list of varres records.
   implicit none
   logical reserved
```

8.8 List things

The routines in this section are intended for the line oriented user interface of GTP. It lists data assuming 80 column width of the screen. In some cases a character variable is returned but in most case the list unit is provided in the call. This can be the screen, a file or a device.

Some listings are described in connection with the objects that are listed, see 8.15.9.

8.8.1 List data for all elements

The element data is listed. Second version for TDB files.

```
 subroutine list_all_elements(unit)
! lists elements
   implicit none
   integer unit
 subroutine list_all_elements2(unit)
! lists elements
   implicit none
   integer unit
```

8.8.2 List data for all components

The components may be different in each equilibrium.

```
 subroutine list_all_components(unit,ceq)
! lists the components for an equilibrium
   implicit none
   integer unit
   TYPE(gtp_equilibrium_data), pointer :: ceq
```

8.8.3 List data for one element

The data for element "elno" in written to the character variable text from position ipos.

```
 subroutine list_element_data(text,ipos,elno)
   implicit none
   character text*(*)
   integer ipos,elno
```

8.8.4 List data for one species

The data for species "spno" in written to the character variable text from position ipos. The second version is suitable for TDB files.

```
 subroutine list_species_data(text,ipos,spno)
   implicit none
   character text*(*)
   integer ipos,spno
 subroutine list_species_data2(text,ipos,loksp)
! loksp is species record ...
   implicit none
   character text*(*)
   integer ipos,loksp
```

8.8.5 List data for all species

One line for each species is listed on device unit.

```
 subroutine list_all_species(unit)
   implicit none
   integer unit
```

8.8.6 List sorted phases

The phases are listed with one line each, first the stable phases, then the entered but unstable in decreasing order of stability. If more than 10 phases the remaining are merged into a single line.

```
 subroutine list_sorted_phases(unit,ceq)
! short list with one line for each phase
! suspended phases merged into one line
! stable first, then entered ordered in driving force order, then dormat
! also in driving force order.  Only 10 of each, the others lumped together
   implicit none
   integer unit
   TYPE(gtp_equilibrium_data), pointer :: ceq
```

8.8.7 List a little data for all phases

One line for each phase is listed on device unit for equilibrium ceq.

```
 subroutine list_all_phases(unit,ceq)
! short list with one line for each phase
! suspended phases merged into one line
   implicit none
   integer unit
   TYPE(gtp_equilibrium_data), pointer :: ceq
```

8.8.8 List global results

This is part of the "list_result" command in the GTP user i/f.

```
 subroutine list_global_results(lut,ceq)
! list G, T, P, V and some other things
   implicit none
   integer lut
   TYPE(gtp_equilibrium_data), pointer :: ceq
```

8.8.9 List components result

This is part of the "list_result" command in the GTP user i/f.

```
 subroutine list_components_result(lut,mode,ceq)
! list one line per component (name, moles, x/w-frac, chem.pot. reference state
! mode 1=mole fractions, 2=mass fractions
   implicit none
   integer lut,mode
   TYPE(gtp_equilibrium_data), pointer :: ceq
```

8.8.10 List all phases with positive dgm

This is part of the "list_result" command in the GTP user i/f. If a phase has positive DGM it should either be dormant or there has been an error calculating the equilibrium.

```
 subroutine list_phases_with_positive_dgm(mode,lut,ceq)
! list one line for each phase+comp.set with positive dgm on device lut
! The phases must be dormant or the result is in error.  mode is not used
   implicit none
   integer mode,lut
   TYPE(gtp_equilibrium_data), pointer :: ceq
```

8.8.11 List results for one phase

This is part of the "list_result" command in the GTP user i/f. It lists normally only the stable phases with their amounts and compositions. With different values of mode units and listing can be changed.

```
 subroutine list_phase_results(iph,jcs,mode,lut,once,ceq)
! list results for a phase+comp.set on lut
! mode specifies the type and amount of results,
! unit digit:   0=mole fraction,      othewise mass fractions
```

```
! 10th digit:   0=only composition,   10=also constitution
! 100th digit:  0=value order,        100=alphabetical order
! 1000th digit: 0=all phases,         1000=only stable phases
   implicit none
   integer iph,jcs,mode,lut
   logical once
   TYPE(gtp_equilibrium_data), pointer :: ceq
```

8.8.12 Format output for constitution

This subroutine formats the output of composition or constitution in nice columns trying to use as few lines as possible.

```
 subroutine format_phase_composition(mode,nv,consts,vals,lut)
! list composition/constitution in alphabetical or value order
! entalsiffra 0 mole fraction, 1 mass fraction, 3 mole percent, 4 mass percent
! tiotalsiffra alphabetical order ... ??
! mode >100 else alphabetical order
! nv is number of components/constitunents (in alphabetical order in consts)
! components/constituents in consts, fractions in vals
   implicit none
   integer nv,mode,lut
   character consts(nv)*(*)
   double precision vals(nv)
```

8.8.13 List data on SCREEN or TDB, LaTeX or macro format

This subroutines list the model parameters for all phases in a selected format. Only SCREEN and TDB are implemented.

```
 subroutine list_many_formats(cline,last,ftyp,unit1)
! lists all data in different formats: SCREEN/TDB/MACRO/LaTeX/ODB
!                                        1    2    3     4    5
! unfinished
   implicit none
   character cline*(*)
   integer last,unit1,ftyp
```

8.8.14 List some phase model stuff

This is probably redundant but can be used to check the conversion from site fractions to disordered fractions for phases with several fraction sets.

```
 subroutine list_phase_model(iph,ics,lut,ceq)
```

```
! list model (no parameters) for a phase on lut
   implicit none
   integer iph,ics,lut
   TYPE(gtp_equilibrium_data), pointer :: ceq
```

8.8.15 List all parameter data for a phase

This is the big listing of the model and data for a phase. It lists the sublattices, sites, constituents. Then all endmembers and all interaction parameters.

The second version is suitable for TDB files.

```
 subroutine list_phase_data(iph,lut)
! list parameter data for a phase on unit lut
   implicit none
   integer iph,lut
 subroutine list_phase_data2(iph,ftyp,ch1,lut)
! list parameter data for a phase on unit lut in ftyp format, ftyp=2 is TDB
   implicit none
   integer iph,lut,ftyp
   character ch1*1
```

8.8.16 Format expression of references for endmembers

When listing an endmember parameter for the Gibbs energy this subroutine subtracts the H298 expression.

```
 subroutine subrefstates(funexpr,jp,lokph,parlist,endm,noelin1)
! list a sum of reference states for a G parameter
! like "-H298(BCC_A2,FE)-3*H298(GRAPITE,C)"
   implicit none
   integer jp,lokph,parlist,endm(*)
   character funexpr*(*)
   logical noelin1
```

8.8.17 Encode stoichiometry of species

This subroutine generate a stoichiometric formula for a species including a charge.

```
 subroutine encode_stoik(text,ipos,spno)
! generate a stoichiometric formula of species from element list
   implicit none
   integer ipos,spno
   character text*(*)
```

8.8.18 Decode stoichiometry of species

This subroutine can translate a stoichiometric formula to elements and stoichiometric factors including a charge.

```
 subroutine decode_stoik(name,noelx,elsyms,stoik)
! decode a species stoichiometry in name to element index and stoichiometry
! all in upper case
   implicit none
   character name*(*),elsyms(*)*2
   double precision stoik(*)
   integer noelx
```

8.8.19 Encode constituent array for parameters

This subroutine generates a constituent array for a parameter. Constituents are species. Constituents in different sublattices are separated by ":", interacting constituents in same sublattice are separated by ",". The degree is written after a ";".

```
 subroutine encode_constarr(constarr,nsl,endm,nint,lint,ideg)
! creates a constituent array
   implicit none
   character constarr*(*)
   integer, dimension(*) :: endm
   integer nsl,nint,ideg
   integer, dimension(2,*) :: lint
```

8.8.20 Decode constituent array for parameters

By providing the indices of constituents in the endmember and possible interaction constituents and the degree, a text with the constitutent array is generated.

```
 subroutine decode_constarr(lokph,constarr,nsl,endm,nint,lint,ideg)
! deconde a text string with a constituent array
! a constituent array has <species> separated by , or : and ; before degree
   implicit none
   character constarr*(*)
   integer endm(*),lint(2,*)
   integer nsl,nint,ideg,lokph,lord
```

8.8.21 List parameter data references

This subroutine lists the source of one or several bibliographic references for the parameters.

```
 subroutine list_bibliography(bibid,lut)
! list bibliographic references
   implicit none
   integer lut
   character bibid*(*)
```

8.8.22 List conditions on a file or screen

The heading says all.

```
 subroutine list_conditions(lut,ceq)
! lists conditions on lut
   implicit none
   integer lut
   type(gtp_equilibrium_data), pointer :: ceq
```

8.8.23 Extract one conditions in a character veriable

A single condition is written in a character variable

```
 subroutine get_one_condition(ip,text,seqz,ceq)
! list the condition with the index seqz into text
! It lists also fix phases and conditions that are not active
   implicit none
   integer ip,seqz
   character text*(*)
   TYPE(gtp_equilibrium_data), pointer :: ceq
```

8.8.24 List experiments

All experimental information associated with the equilibrium ceq is listed on unit lut. The get_one_experiment routine returns the text.

```
 subroutine list_experiments(lut,ceq)
! list all experiments into text
   implicit none
   integer lut
   TYPE(gtp_equilibrium_data), pointer :: ceq
 subroutine get_one_experiment(ip,text,seqz,ceq)
! list the experiment with the index seqz into text
! It lists also experiments that are not active ??
! UNFINISHED current value should be appended
   implicit none
```

```
   integer ip,seqz
   character text*(*)
   TYPE(gtp_equilibrium_data), pointer :: ceq
```

8.8.25 List condition in character variable

All current active conditions in equilibrium ceq is written to the character variable text. This can be written on the screen or used for other purposes. It can also be used for experiments (not implemented yet).

```
 subroutine get_all_conditions(text,mode,ceq)
! list all conditions if mode=0, experiments if mode=1
   implicit none
   integer mode
   character text*(*)
   TYPE(gtp_equilibrium_data), pointer :: ceq
```

8.8.26 List available parameter identifiers

The GTP package allows definition of new properties that can be modeled as dependent on the constitution of each phase. Such properties must be defined in the software and they can be listed with this subroutine. Many additions depend on such parameter properties like the Curie temperature and the Debye temperature.

On can also add properties that does not affect the Gibbs energy but which depend on the constitution of the phase like the mobility, resistivity, lattice parameter etc.

```
 subroutine list_defined_properties(lut)
! lists all parameter identifiers allowed
   implicit none
   integer lut
```

8.8.27 Find defined properties

Although properties like TC (Curie temperature) and BMAG (Average Bohr magneton number) are not state variables they can be listed using the command LIST STATE_VARIABLEs and their values can be obtained by the same subroutines that are used for state variables like get_state_variable. They use the following subroutine to find the properties defined in the gtp_propid structure.

```
 subroutine find_defined_property(symbol,mode,typty,iph,ics)
! searches the propid list for one with symbol or identifiction typty
! if mode=0 then symbol given, if mode=1 then typty given
! symbol can be TC(BCC), BM(FCC), MQ&FE(HCP) etc, the phase must be
```

```
! given in symbol as otherwise it is impossible to find the constituent!!!
! A constituent may have a sublattice specifier, MQ&FE#3(SIGMA)
   implicit none
   integer mode,typty,iph,ics
   character symbol*(*)
```

8.8.28 List some odd details

I do not remember what this is used for.

```
 subroutine list_equilibria_details(mode,teq)
! not used yet ...
   implicit none
   TYPE(gtp_equilibrium_data), pointer :: teq
   integer mode
```

8.8.29 List an error message if any

I do not remember what this is used for.

```
 logical function gtp_error_message(reset)
! tests the error code and writes the error message (if any)
! and reset error code if reset=0
! if reset >0 that is set as new error message
! if reset <0 the error code is not changed
! return TRUE if error code set, FALSE if error code is zero
   implicit none
   integer reset
```

8.9 Interactive subroutines

The current user interface to OC and GTP is command oriented and there are subroutines provided in GTP to enter, set, list and get many things. Most subroutines where the user is expected to provide information is collected in this section.

8.9.1 Ask for phase constitution

The used can provide the default constitution or enter a constitution specificly for a phase and composition set.

```
 subroutine ask_phase_constitution(cline,last,iph,ics,lokcs,ceq)
! interactive input of a constitution of phase iph
   implicit none
```

```
   integer last,iph,ics,lokcs
   character cline*(*)
 subroutine ask_phase_new_constitution(cline,last,iph,ics,lokcs,ceq)
! interactive input of a constitution of phase iph
   implicit none
   integer last,iph,ics,lokcs
   character cline*(*)
```

8.9.2 Ask for parameter

The user can enter a model parameter with this subroutine.

```
 subroutine enter_parameter_interactivly(cline,ip,mode)
! enter a parameter from terminal or macro
! NOTE both for ordered and disordered fraction set !!
! mode = 0 for entering
!        1 for listing on screen (kou)
   implicit none
   integer ip,mode
   character cline*(*)
```

8.9.3 Amend global bits

There are a number of global bits that can be set by this subroutine.

```
 subroutine amend_global_data(cline,ipos)
   implicit none
   character cline*(*)
   integer ipos
```

8.9.4 Ask for reference of parameter data

Each parameter in a model can have a data reference, preferably a published paper. When a parameters is entered by calling enter_parameter_interactivly the reference is asked for but with this routine it is possible to enter such a reference separately.

```
 subroutine enter_reference_interactivly(cline,last,mode,iref)
! enter a reference for a parameter interactivly
! this should be modified to allow amending an existing reference
   implicit none
   character cline*(*)
   integer last,mode,iref
```

8.9.5 Enter an experiment

With this subroutine the state variable for an experiment, its value and uncertainty is given for equilibrium ceq. The experiment can also be changed or removed (set equal to NONE).

The logical function is used to check if two state variable records represent the same state variable (because associated would only be true if they were the same record).

```
 subroutine enter_experiment(cline,ip,ceq)
! enters an experiment, almost the same as set_condition
   implicit none
   character cline*(*)
   integer ip
   type(gtp_equilibrium_data), pointer :: ceq
 logical function same_statevariable(svr1,svr2)
! returns TRUE if the state variable records are identical
   type(gtp_state_variable), pointer :: svr1,svr2
```

8.9.6 Set a condition

This is the central routine to set a condition for an equilibrium calculation. Anther alternative is the set_input_amount. When setting the status of a phase as fixed this subroutine is called automatically to add this as condition.

```
 subroutine set_condition(cline,ip,ceq)
! to set a condition
   implicit none
   character cline*(*)
   integer ip
   type(gtp_equilibrium_data), pointer :: ceq
 subroutine set_cond_or_exp(cline,ip,new,notcond,ceq)
! decode an equilibrium condition, can be an expression with + and -
! the expression should be terminated with an = or value supplied on next line
! like "T=1000", "x(liq,s)-x(pyrrh,s)=0", "mu(cr)-1.5*mu(o)=muval"
! Illegal with number before first state variable !!!
! It can also be a "NOFIX=<phase>" or "FIX=<phase> value"
! The routine should also accept conditions identified with the "<number>:"
! where <number> is that preceeding each condition in a list_condition
! It should also accept changing conditions by <number>:=new_value
! The pointer to the (most recent) condition or experiment is returned in new
! notcond is 0 if a condition should be created, otherwise an experiment
   implicit none
   integer ip,notcond
   character cline*(*)
   TYPE(gtp_equilibrium_data), pointer :: ceq
```

```
   TYPE(gtp_condition), pointer :: new
 subroutine get_experiment_with_symbol(symsym,experimenttype,temp)
! finds an experiment with s symbol index symsym and exp.type
   implicit none
   integer symsym,experimenttype
   type(gtp_condition), pointer :: temp
! NOTE: temp must have been set to ceq%lastcondition before calling this
```

8.9.7 Get condition record

A condition is typically a state variable assigned a value like $T = 1273$ but it can be much more complicated. One can have conditions like $x(\text{liquid}, \text{S}) - x(\text{pyrrhotite}, \text{S}) = 0$ to specify the congruent melting point of pyrrhotite. A condition can also be that a phase is fixed, i.e. prescribed to be stable.

A condition is specified by the number of terms, the state variable with possible indices (to specify phase, component or constituent etc), reference state (for chemical potentials) and unit (Joule or calorie or per cent or fraction)

The state variable record was not used when this subroutine was first written and the original version is now called get_condition2. This version is depreciated and should not be used. The subroutine that replaces this has the original name and has a state variable record as argument. An additional subroutine that returns the state variable record is also available.

The first two subroutines return a pointer to the condition record, see gtp_condition for that structure, or an error code if no such condition.

The subroutine to set conditions is decribed in 8.9.6.

```
 subroutine get_condition(nterm,svr,pcond)
! finds a condition/experiment record with the given state variable expression
! If nterm<0 svr is irrelevant, the absolute value of nterm is the sequential
! number of the ACTIVE conditions
   implicit none
   integer nterm
   type(gtp_state_variable), pointer :: svr
! NOTE: pcond must have been set to ceq%lastcondition before calling this
! pcond: pointer, to a gtp_condition record for this equilibrium
   type(gtp_condition), pointer :: pcond
 subroutine get_condition2(nterm,coeffs,istv,indices,iref,iunit,pcond)
! finds a condition record with the given state variable expression
! nterm: integer, number of terms in the condition expression
! istv: integer, state variable used in the condition
! indices: 2D integer array, state variable indices used in the condition
! iref: integer, reference state of the condition (if applicable)
! iunit: integer, unit of the condition value
! NOTE: pcond must have been set to ceq%lastcond before calling this routine!!!
```

```
! pcond: pointer, to a gtp_condition record for this equilibrium
! NOTE: conditions like expressions x(mg)-2*x(si)=0 not implemeneted
! fix phases as conditions have negative condition variable
   implicit none
   TYPE(gtp_condition), pointer :: pcond
   integer, dimension(4,*) :: indices
   integer nterm,istv,iref,iunit
   double precision coeffs(*)
 subroutine extract_stvr_of_condition(pcond,nterm)
! finds a condition record with the given state variable record
! returns it as a state variable record !!!
! nterm: integer, number of terms in the condition expression
! pcond: pointer, to a gtp_condition record
   implicit none
   TYPE(gtp_condition), pointer :: pcond
   integer nterm
```

8.9.8 A utility routine to locate a condition record

This is needed during STEP and MAP to find an axis condition.

```
 subroutine locate_condition(seqz,pcond,ceq)
! locate a condition using a sequential number
   implicit none
   integer seqz
   type(gtp_condition), pointer :: pcond
   type(gtp_equilibrium_data), pointer :: ceq
```

8.9.9 A utility routine to get the current value of a condition

This subroutine is called at each iteration in the equilibrium calculation to formulate the system matrix containing the external conditions. It is still very rudimentary and can be improved.

We do not need to pass the pointer to ceq as that is found via the condition record.

```
 subroutine apply_condition_value(current,what,value,cmix,ccf,ceq)
! This is called when calculating an equilibrium.
! It returns a condition at each call, at first call current must be nullified?
! When all conditions done the current is nullified again
! If what=-1 then return degrees of freedoms and maybe something more
! what=0 means calculate current values of conditions
! calculate the value of a condition, used in minimizing G
! ccf are the coefficients for conditions with several terms
   implicit none
```

```
   integer what,cmix(*)
   double precision value,ccf(*)
   TYPE(gtp_equilibrium_data), pointer :: ceq
   TYPE(gtp_condition), pointer :: current
 subroutine condition_value(mode,pcond,value,ceq)
! set (mode=0) or get (mode=1) a new value of a condition.  Used in mapping
   implicit none
   integer mode
   type(gtp_condition), pointer :: pcond
   type(gtp_equilibrium_data), pointer :: ceq
   double precision value
```

8.9.10 Ask for new set of components

When working with real quasi-binary or quasi-ternary systems when the models of the phases does not extend outside this system, like the CaO-SiO_2 system, it is convenient to specify CaO, SiO_2 as components rather than the elements Ca, Si and O. However, one cannot change the number of components this way, one must also specify a third element like CaO, SiO2, O. The condition for the third component can be removed by giving an arbitrary chemical potential or activity.

If one defines a new set of components can calculates a composition outside the hyper-volume defined by these components one may have negative fractions of the components.

This subroutine is not implemented yet.

```
 subroutine amend_components(cline,last,ceq)
! enter a new set of components for equilibrium ceq
   implicit none
   integer last
   character cline*(*)
   type(gtp_equilibrium_data), pointer :: ceq
```

8.9.11 Ask for default phase constitution

The user can enter a default constitution for a phase. A negative value means a maximum value, a positive means a minimum value.

```
 subroutine ask_default_constitution(cline,last,iph,ics,ceq)
! set values of default constitution interactivly
! phase and composition set already given
   implicit none
   character cline*(*)
   integer last,iph,ics
   TYPE(gtp_equilibrium_data), pointer :: ceq
```

8.9.12 Set default constitution

The constitution of a phase is set to its default constitution

```
 subroutine enter_default_constitution(iph,ics,mmyfr,ceq)
! user specification of default constitution for a composition set
   implicit none
   TYPE(gtp_equilibrium_data), pointer :: ceq
   integer iph,ics
   real mmyfr(*)
```

8.9.13 Interactive set input amounts

This subroutine allows setting condition by entering the amount of species. The amount of the species is converted to amount of components internally. Redundancy is allowed. If several species contain the same element the amounts are added. For example set_input_amount N(C1O1)=10, N(H2O1)=5, N(C1H4)=7, N(O2)=20 is translated to the conditions N(C)=17, N(H)=38, N(O)=55.

```
 subroutine set_input_amounts(cline,lpos,ceq)
! set amounts like n(specie)=value or b(specie)=value
! value can be negative removing amounts
! values are converted to moles and set or added to conditions
   implicit none
   integer lpos
   character cline*(*)
   TYPE(gtp_equilibrium_data), pointer :: ceq
```

8.9.14 Utility to decode a parameter identifier

Used when entering data

```
 subroutine get_parameter_typty(name1,lokph,typty,fractyp)
! interpret parameter identifiers like MQ&C#2 in MQ&C#2(FCC_A1,FE:C) ...
! find the property associated with this symbol
   integer typty,fractyp,lokph
   character name1*(*)
```

8.10 Save and read data from files

Most of the subroutines in this section are unfinished. The only thing that can be read is a simple TDB file.

The problem is that whenever a change is made in the data structure these routines must be modified accordingly. And the data structure is still undergoing big changes.

These subroutines are in total disorder

8.10.1 Save all data

Not implemented yet.

```
 subroutine gtpsave(filename,str)
! save all data on file, unformatted, TDB or macro
! header
! element list
! species list
! phase list with sublattices, endmembers, interactions and parameters etc
! tpfuns
! state variable functions
! references
!
   implicit none
   character*(*) filename,str
```

8.10.2 Save data in LaTeX and other formats

This is just a dream.

```
 subroutine gtpsavelatex(filename,specification)
! save all data on LaTeX format on a file (for publishing)
! header
! element list
! species list
! phase list with sublattices, endmembers, interactions and parameters etc
! tpfuns
! state variable functions
! references
! equilibrium record(s) with conditions, componenets, phase_varres records etc
! anything else?
   implicit none
   character*(*) filename,specification
 subroutine gtpsavedir(filename,specification)
! save all data on a direct file (random access)
! header
! element list
! species list
! phase list with sublattices, endmembers, interactions and parameters etc
```

```
! tpfuns
! state variable functions
! references
! equilibrium record(s) with conditions, componenets, phase_varres records etc
! anything else?
   implicit none
   character*(*) filename,specification
 subroutine gtpsavetdb(filename,specification)
! save all data in TDB format on an file
! header
! element list
! species list
! phase list with sublattices, endmembers, interactions and parameters etc
! tpfuns
! state variable functions
! references
! equilibrium record(s) with conditions, componenets, phase_varres records etc
! anything else?
   implicit none
   character*(*) filename,specification
```

8.10.3 Save all data again

This is complicated.

```
 subroutine gtpsaveu(filename,specification)
! save all data unformatted on an file
! header
! element list
! species list
! phase list with sublattices, endmembers, interactions and parameters etc
! tpfuns
! state variable functions
! references
! equilibrium record(s) with conditions, componenets, phase_varres records etc
! anything else?
   implicit none
   character*(*) filename,specification
```

8.10.4 Save data for a phase

The title says all

```
 subroutine savephase(lut,lokph)
```

```
! save data for phase at location lokph (except data in the equilibrium record)
! For phases with disordered set of parameters we must access the number of
! sublattices via firsteq
   implicit none
   integer lut,lokph
```

8.10.5 Save data for an equilibrium record

The title says all

```
 subroutine saveequil(lut,ceq)
! save data for an equilibrium record
   implicit none
   integer lut
   type(gtp_equilibrium_data), pointer :: ceq
```

8.10.6 Save the state variable functions on file

```
 subroutine svfunsave(lut,ceq)
! saves all state variable functions on a file
   implicit none
   integer lut
   type(gtp_equilibrium_data), pointer :: ceq
```

8.10.7 Save the bibliographic references on file

```
 subroutine bibliosave(lut)
! saves references on a file
   implicit none
   integer lut
```

8.10.8 Read data from a saved file

Not implemented yet.

```
 subroutine gtpread(filename,str)
! read unformatted all data in the following order
! header
! element list
! species list
! phase list with sublattices, endmembers, interactions and parameters etc
! tpfuns
! state variable functions
```

```
! references
! equilibrium record(s) with conditions, componenets, phase_varres records etc
!
   implicit none
   character*(*) filename,str
```

8.10.9 Reading unformatted data for a phase

```
 subroutine readphase(lin,jdum)
! read data for phlista and all endmembers etc
! works for test case without disordered fraction test
   implicit none
   integer lin,jdum
```

8.10.10 Read unformatted data for an endmember

```
 subroutine readendmem(lin,nsl,emrec,nop,noi,nem)
! allocates and reads an endmember record
   implicit none
   integer lin,nsl,nop,noi,nem
   type(gtp_endmember), pointer :: emrec
```

8.10.11 Read unformatted data for a property

```
 subroutine readproprec(lin,proprec,nox)
! allocates and reads a property record
   implicit none
   integer lin,nox
   type(gtp_property), pointer :: proprec
```

8.10.12 Read unformatted data for an interaction

```
 subroutine readintrec(lin,intrec,mult,noi,nup,nop)
! allocates and reads an interaction record UNFINISHED
   implicit none
   integer lin,mult,noi,nup,nop
   type(gtp_interaction), pointer :: intrec
```

8.10.13 Read unformatted data for an equilibrium

```
 subroutine readequil(lin,ceq)
! Read equilibria records from a file
   implicit none
```

```
   integer lin
   type(gtp_equilibrium_data), pointer :: ceq
```

8.10.14 Read state variable functions

```
 subroutine svfunread(lin)
! read a state variable function from save file and store it.
! by default there are some state variable functions, make sure
! they are deleted.  Done here just by setting nsvfun=0
   implicit none
   integer lin
```

8.10.15 Read reference records

```
 subroutine biblioread(lin)
! read references from save file
   implicit none
   integer lin
```

8.10.16 Erase all data

This is necessary before reading a new saved file. Not implemented yet.

```
 subroutine new_gtp
!
! DELETES ALL DATA so a new TDB file can be read
!
! this is needed before reading a new unformatted file (or same file again)
! we must go through all records and delete and deallocate each
! separately.  Very similar to gtpread
   implicit none
```

8.10.17 Delete a phase

```
 subroutine delphase(lokph)
! save data for phase at location lokph (except data in the equilibrium record)
! For phases with disordered set of parameters we must access the number of
! sublattices via firsteq
   implicit none
   integer lokph
```

8.10.18 Yet another utility routine

```
 logical function iskeyword(text,keyword,nextc)
! compare a text with a given keyword. Abbreviations allowed
! but the keyword and abbreviation must be surrounded by spaces
! nextc set to space character in text after the (abbreviated) keyword
   implicit none
   character text*(*),keyword*(*),key*64
   integer nextc
```

8.10.19 And some more utility routines

Detects a TDB keyword and replaces TAB character by a space.

```
 integer function istdbkeyword(text,nextc)
! compare a text with a given keyword. Abbreviations allowed (not within _)
! but the keyword and abbreviation must be surrounded by spaces
! nextc set to space character in text after the (abbreviated) keyword
   implicit none
   character text*(*)
   integer nextc
 subroutine replacetab(line,nl)
! replaces TAB by space in line
   implicit none
   character line*(*)
   integer nl
```

8.10.20 Read a TDB file

This subroutine can read a TDB file that is not too fancily edited manually. Best is to read as written from Thermo-Calc. Some TYPE_DEFINITIONS are not handelled, especially the DISORDERED_PART as this has been implemented differently in GTP.

```
 subroutine readtdb(filename,nel,selel)
! reading data from a TDB file with selection of elements
!-------------------------------------------------------
! Not all TYPE_DEFS implemented
!-------------------------------------------------------
   implicit none
   integer nel
   character filename*(*),selel(*)*2
```

8.10.21 Check a TDB file exists and extract elements

This is used to check a user typed a correct TDB file name and extracts the elements so the user can select which he wants.

```
 subroutine checktdb(filename,nel,selel)
! checking a TDB file exists and return the elements
   implicit none
   integer nel
   character filename*(*),selel(*)*2
```

8.11 State variable stuff

State variables are important for the setting and extracting results of a calculation. State variables are treated very similarly to Thermo-Calc using symbols like T, P, N, $x(<$ component $>$) etc.

The internal syntax of state variables is rather complicated, perhaps it should be revised and defined as a structure? If there are errors or one wants to make modifications it is not easy.

Things like Curie temperature, Debye temperature, mobilities etc are tread almost like "state variables" although one cannot use them in conditions. Adding more things like elastic constants will be a bit complicated.

The subroutines for manipulations is also a bit complicated and could do with a clean up and renaming.

8.11.1 Get state variable value given its symbol

By providing a state variable as a character variable like T or $x(\text{liquid}, \text{cr})$ this routine returns its current value. Wildcards, "*", are not allowed, see 8.11.2.

A variant checks if a specified phase is stable.

```
 subroutine get_stable_state_var_value(statevar,value,encoded,ceq)
! called with a state variable character
! If the state variable includes a phase it checks if the phase is stable ...
   implicit none
   TYPE(gtp_equilibrium_data), pointer :: ceq
   character statevar*(*),encoded*(*)
   double precision value
 subroutine get_state_var_value(statevar,value,encoded,ceq)
! called with a state variable character
   implicit none
   TYPE(gtp_equilibrium_data), pointer :: ceq
```

```
   character statevar*(*),encoded*(*)
   double precision value
```

8.11.2 Get many state variable values

This routine can be called with wildcard, "*", as argument in state variables like $NP(*)$, $x(*,CR)$ etc. It is fragile and currently only available when defining plot axis and some output.

```
 subroutine get_many_svar(statevar,values,mjj,kjj,encoded,ceq)
! called with a state variable name with woldcards allowed like NP(*), X(*,CR)
! mjj is dimension of values, kjj is number of values returned
! encoded used to specify if phase data in phasetuple order ('Z')
! >>>> BIG problem: How to do with phases that are note stable?
! If I ask for w(*,Cr) I only want the fraction in stable phases
! but whenthis is used for GNUPLOT the values are written in a matix
! and the same column in that phase must be the same phase ...
! so I have to have the same number of phases from each equilibria.
!
   implicit none
   TYPE(gtp_equilibrium_data), pointer :: ceq
   character statevar*(*),encoded*(*)
   double precision values(*)
   integer mjj,kjj
```

8.11.3 Decode a state variable symbol

This subroutine takes as input a character with a state variable and returns a state variable record with its specification. It can also handle decoding of property parameters symbols like the Curie temperature. The main routine calls the older version of this subroutine with a more complex handling of state variables. This second subroutine will eventually disappear and should not be used.

The new version of this subroutine calls the old but this will be removed in a future release. If there are any changes in the state variables structure several subroutines must be changed like this one, 8.11.8 and 8.11.10.

The subroutine also handle property symbols used in the parameters, see 2.3.3, to make it possible to obtain the value of such a propery after an equilibrium calculation. The value returned in svrindex.

```
 subroutine decode_state_variable(statevar,svr,ceq)
! converts a state variable character to state variable record
   character statevar*(*)
   type(gtp_state_variable), pointer :: svr
   type(gtp_equilibrium_data), pointer :: ceq
```

```
! this subroutine using state variable records is a front end of the next:
 subroutine decode_state_variable3(statevar,istv,indices,iref,iunit,svr,ceq)
! converts an old state variable character to indices
! Typically: T, x(fe), x(fcc,fe), np(fcc), y(fcc,c#2), ac(h2,bcc), ac(fe)
! NOTE! model properties like TC(FCC),MQ&FE(FCC,CR) must be detected
! NOTE: added storing information in a gtp_state_variable record svrec !!
!
! this routine became as messy as I tried to avoid
! but I leave it to someone else to clean it up ...
!
! state variable and indices
! Symbol  no   index1 index2 index3 index4
! T       1    -
! P       2    -
! MU      3    component or phase,constituent
! AC      4    component or phase,constituent
! LNAC    5    component or phase,constituent
!                                          index (in svid array)
! U       10   (phase#set)                    6     Internal energy (J)
! UM      11    "                             6     per mole components
! UW      12    "                             6     per kg
! UV      13    "                             6     per m3
! UF      14    "                             6     per formula unit
! S       2x    "                             7     entropy
! V       3x    "                             8     volume
! H       4x    "                             9     enthalpy
! A       5x    "                            10     Helmholtz energy
! G       6x    "                            11     Gibbs energy
! NP      7x    "                            12     moles of phase
! BP      8x    "                            13     mass of moles
! DG      9x    "                            15     Driving force
! Q       10x   "                            14     Internal stability
! N       11x (component/phase#set,component) 16  moles of components
! X       111   "                            17     mole fraction of components
! B       12x   "                            18     mass of components
! W       122   "                            19     mass fraction of components
! Y       13    phase#set,constituent#subl   20     constituent fraction
!----- model variables <<<< these now treated differently
! TC      -     phase#set                    -      Magnetic ordering T
! BMAG    -     phase#set                    -      Aver. Bohr magneton number
! MQ&     -     element, phase#set           -      Mobility
! THET    -     phase#set                    -      Debye temperature
!
   implicit none
   integer, parameter :: noos=20
   character*4, dimension(noos), parameter :: svid = &
```

```
      ['T   ','P   ','MU  ','AC  ','LNAC','U   ','S   ','V   ',&
       'H   ','A   ','G   ','NP  ','BP  ','DG  ','Q   ','N   ',&
       'X   ','B   ','W   ','Y   ']
!       1     2     3     4     4     6     7     8
   character statevar*(*)
   integer istv,iref,iunit
   integer, dimension(4) :: indices
   type(gtp_equilibrium_data), pointer :: ceq
! I shall try to use this record type instead of separate arguments: !!
!   type(gtp_state_variable), pointer :: svrec
   type(gtp_state_variable), pointer :: svr
```

8.11.4 Calculate molar and mass properties for a phase

This subroutine calculates mole and mass fractions of all components for a phase (mole fractions of components not dissolved is zero). It also returns the total number of moles of components and the mass. In amount the number of moles per formula unit is returned (same as qq(1) in get_phase_data and set_constitution).

```
 subroutine calc_phase_molmass(iph,ics,xmol,wmass,totmol,totmass,amount,ceq)
! calculates mole fractions and mass fractions for a phase#set
! xmol and wmass are fractions of components in mol or mass
! totmol is total number of moles and totmass total mass of components.
! amount is number of moles of components per formula unit.
   implicit none
   TYPE(gtp_equilibrium_data) :: ceq
   integer iph,ics
   double precision, dimension(*) :: xmol,wmass
   double precision amount,totmol,totmass
```

8.11.5 Calculate molar amounts for a phase

Specially used for grid minimization.

```
 subroutine calc_phase_mol(iph,xmol,ceq)
! calculates mole fractions for phase iph, compset 1 in equilibrium ceq
! used for grid generation and some other things
! returns current constitution in xmol equal to mole fractions of components
   implicit none
   integer iph
   double precision xmol(*)
   TYPE(gtp_equilibrium_data) :: ceq
```

8.11.6 Sum molar and mass properties for all phases

Sums the mole and mass fractions for all components and also total number of moles and mass over all stable phases using 8.11.4.

```
 subroutine calc_molmass(xmol,wmass,totmol,totmass,ceq)
! summing up N and B for each component over all phases with positive amount
! Check that totmol and totmass are correct ....
   implicit none
   double precision, dimension(*) :: xmol,wmass
   double precision totmol,totmass
   TYPE(gtp_equilibrium_data) :: ceq
```

8.11.7 Sum all normalizing property values

Used to calculate normalizing propertes like V, N and B but also G and S for the whole system. Used when calculating state variable values.

```
 subroutine sumprops(props,ceq)
! summing up G, S, V, N and B for all phases with positive amount
! Check if this is correct
   implicit none
   TYPE(gtp_equilibrium_data) :: ceq
   double precision props(5)
```

8.11.8 Encode state variable

This converts the internal coding of a state variable into a character variable text starting at position ip. ip is updated inside.

The subroutine also handle property symbols used in the parameters, see 2.3.3, to make it possible to list the symbol of such a propery after an equilibrium calculation. See 8.11.3

The new version of this subroutine uses state variable records, the previous one using individual arguments should not be used as it will eventually disappear.

```
 subroutine encode_state_variable(text,ip,svr,ceq)
! writes a state variable in text form position ip.  ip is updated
   character text*(*)
   integer ip
   type(gtp_state_variable), pointer :: svr
   type(gtp_equilibrium_data), pointer :: ceq
 subroutine encode_state_variable3(text,ip,istv,indices,iunit,iref,ceq)
! writes a state variable in text form position ip.  ip is updated
! the internal coding provides in istv, indices, iunit and iref
```

```
! ceq is needed as compopnents can be different in different equilibria ??
! >>>> unfinished as iunit and iref not really cared for
   implicit none
   integer, parameter :: noos=20
   character*4, dimension(noos), parameter :: svid = &
       ['T   ','P   ','MU  ','AC  ','LNAC','U   ','S   ','V   ',&
        'H   ','A   ','G   ','NP  ','BP  ','DG  ','Q   ','N   ',&
        'X   ','B   ','W   ','Y   ']
   character*(*) text
   integer, dimension(4) :: indices
   integer istv,ip,iunit,iref
   type(gtp_equilibrium_data), pointer :: ceq
```

8.11.9 Encode a state variable record

This is provided to convert a single state variable record to a text.

```
 subroutine encode_state_variable_record(text,ip,svr,ceq)
! writes a state variable in text form position ip.  ip is updated
! the svr record provide istv, indices, iunit and iref
! ceq is needed as compopnents can be different in different equilibria ??
! >>>> unfinished as iunit and iref not really cared for
   implicit none
   integer, parameter :: noos=20
   character*4, dimension(noos), parameter :: svid = &
       ['T   ','P   ','MU  ','AC  ','LNAC','U   ','S   ','V   ',&
        'H   ','A   ','G   ','NP  ','BP  ','DG  ','Q   ','N   ',&
        'X   ','B   ','W   ','Y   ']
   character*(*) text
   type(gtp_state_variable) :: svr
   type(gtp_equilibrium_data), pointer :: ceq
```

8.11.10 Calculate state variable value

This is the subroutine that actually calculates the value of a state variable. The state variable is identified using the internal coding.

The subroutine also handle properties used in the parameters, see 2.3.3, to make it possible to obtain the value of such a propery after an equilibrium calculation. The values of istv etc must be as returned from decode_state_variable, see 8.11.3.

```
 subroutine state_variable_val(svr,value,ceq)
! calculate the value of a state variable in equilibrium record ceq
! It transforms svr data to old format and calls state_variable_val3
   type(gtp_state_variable), pointer :: svr
```

```
   double precision value
   TYPE(gtp_equilibrium_data), pointer :: ceq
 subroutine state_variable_val3(istv,indices,iref,iunit,value,ceq)
! calculate the value of a state variable in equilibrium record ceq
! istv is state variable type (integer)
! indices are possible specifiers
! iref indicates use of possible reference state, 0 current, -1 SER
! iunit is unit, (K, oC, J, cal etc). For % it is 100
! value is the calculated values. for state variables with wildcards use
! get_many_svar
   implicit none
   integer, dimension(4) :: indices
   TYPE(gtp_equilibrium_data), pointer :: ceq
   integer istv,iref,iunit
   double precision value
```

8.11.11 The value of the user defined reference state

The value of many state variables depend on the selected reference state. By default that is the value at 298.15 K and 1 bar for the stable phase of the pure elements, this is called SER. If the user defines another reference state this subroutine calculates that value. The reference state is set calling the routine in section 8.6.2.

```
 subroutine calculate_reference_state(kstv,iph,ics,aref,ceq)
! Calculate the user defined reference state for extensive properties
! kstv is the typde of property: 1 U, 2 S, 3 V, 4 H, 5 A, 6 G
! It can be phase specific (iph.ne.0) or global (iph=0)
   implicit none
   integer kstv,iph,ics
   double precision aref
   type(gtp_equilibrium_data), pointer :: ceq
```

8.11.12 A utility ... sort phase in phase tuple order

This is probably redundant

```
 subroutine sortinphtup(n,m,xx)
! subroutine sortinphtup(n,m,xx,ceq)
! subroutine to sort the values in xx which are in phase and compset order
! in phase tuple order.  This is needed by the TQ interface
! The number of values belonging to the phase is m (for example composition)
! argument ceq added as new composition sets can be created ...
   integer n,m
   double precision xx(n*m)
!   type(gtp_equilibrium_data), pointer :: ceq
```

8.12 State variable functions

This section is separated from the state variables itself to make it a little simpler. State variable function can contain any combination of state variables using normal operators like +, -, *, / but also EXP, LN, LOG10, ERF etc. The PUTFUN subroutine in the METLIB package is used. No derivatives can be calculated. A state variable function can refer to another state variable function.

An extension planned but not yet implemented is to allow formal arguments when defining a state variable function, for example CP(@P)=HM(@P).T where the formal argument @P means a phase. @S would stand for a species and @C for a component. When calling the function an actual argument must be supplied.

Another extension that has been partially implemented is the "dot derivatives" meaning the derivative of a state variable with respect to another. This required several changes of the subroutines for state variable functions and some of them have been moved to the minimizer package and are described there.

8.12.1 Enter a state variable function

The first subroutine enters a state variable function and the other stores it in the SVFLISTA array. The old version will eventually disappear.

```
 subroutine enter_svfun(cline,last,ceq)
! enter a state variable function
   implicit none
   integer last
   character cline*(*)
   type(gtp_equilibrium_data), pointer :: ceq
 subroutine set_putfun_constant(svfix,value)
! changes the value of a putfun constant
! svfix is index, value is new value
   implicit none
   integer svfix
   double precision value
 subroutine store_putfun(name,lrot,nsymb,iarr)
! enter an expression of state variables with name name with address lrot
! nsymb is number of formal arguments
! iarr identifies these
! idot if derivative
   implicit none
   character name*(*)
   type(putfun_node), pointer :: lrot
   integer nsymb
   integer iarr(10,*)
 subroutine store_putfun_old(name,lrot,nsymb,&
```

```
       istv,indstv,iref,iunit,idot)
! enter an expression of state variables
! name: character, name of state variable function
! lrot: pointer, to a putfun_node that is the root of the stored expression
! nsymb: integer, number of formal arguments
! istv: integer array, formal argument state variables typ
! indstv: 2D integer array, indices for the formal state variables
! iref: integer array, reference for the formal state variables
! iunit: integer array, unit of the formal state variables
   implicit none
   type(putfun_node), pointer :: lrot
   integer nsymb
   integer, dimension(*) :: istv,iref,iunit,idot
   integer, dimension(4,*) :: indstv
   character name*(*)
```

8.12.2 List a state variable function

These two subroutines can find a state variable function and list its name in a character respectively.

```
 subroutine find_svfun(name,lrot,ceq)
! finds a state variable function called name (no abbreviations)
   implicit none
   character name*(*)
   integer lrot
   type(gtp_equilibrium_data), pointer :: ceq
 subroutine list_svfun(text,ipos,lrot,ceq)
! list a state variable function
   implicit none
   character text*(*)
   integer ipos,lrot
   type(gtp_equilibrium_data), pointer :: ceq
```

8.12.3 Utility subroutine for state variable functions

Utility to store state variable identification for a function.

```
 subroutine make_stvrec(svr,iarr)
! stores appropriate values from a formal argument list to a state variable
! function in a state variable record
   implicit none
   type(gtp_state_variable), pointer :: svr
   integer iarr(10)
```

8.12.4 List all state variable functions

Lists all state variables functions on device kou.

```
 subroutine list_all_svfun(kou,ceq)
! list all state variable funtions
   implicit none
   integer kou
   type(gtp_equilibrium_data), pointer :: ceq
```

8.12.5 Some depreciated routines

The double precision function evaluate_svfun_old(lrot,actual_arg,mode,ceq) is now in the minimizer package.

```
 subroutine evaluate_all_svfun_old(kou,ceq)
! THIS SUBROUTINE MOVED TO MINIMIZER but kept for initiallizing
! cannot be used for state variable functions that are derivatives ...
! evaluate and list values of all functions
   implicit none
   integer kou
   TYPE(gtp_equilibrium_data), pointer :: ceq
 double precision function evaluate_svfun_old(lrot,actual_arg,mode,ceq)
! THIS SUBROUTINE MOVED TO MINIMIZER
! but needed in some cases in this module ... ???
! envaluate all funtions as they may depend on each other
! actual_arg are names of phases, components or species as @Pi, @Ci and @Si
! needed in some deferred formal parameters  (NOT IMPLEMENTED YET)
   implicit none
   integer lrot,mode
   character actual_arg(*)*(*)
   TYPE(gtp_equilibrium_data), pointer :: ceq
```

8.13 Status for things

There are many status bits in various records. These subroutines and function set, clear or test these.

Setting means to set the bit to 1 and clearing means to set the bit to 0. A problem is that I always mix up if BTEST return TRUE or FALSE it the bit is set.

The subroutines and functions in this section should be simplified.

8.13.1 Set status for elements

An element can be entered or suspended.

```
 subroutine change_element_status(elname,nystat,ceq)
! change the status of an element, can affect species and phase status
! nystat:0=entered, 1=suspended, -1 special (exclude from sum of mole fraction)
!
! suspending elements for each equilibrium separately not yet implemented
!
   implicit none
   character elname*(*)
   integer nystat
   TYPE(gtp_equilibrium_data), pointer :: ceq
```

8.13.2 Test status for element

An element can be entered or suspended.

```
 logical function testelstat(iel,status)
! return value of element status bit
   implicit none
   integer iel,status
```

8.13.3 Set status for species

A species can be entered or suspended.

```
 subroutine change_species_status(spname,nystat,ceq)
! change the status of a species, can affect phase status
! nystat:0=entered, 1=suspended
   implicit none
   integer nystat
   character spname*(*)
   TYPE(gtp_equilibrium_data), pointer :: ceq
```

8.13.4 Test status for species

A species can be entered or suspended.

```
 logical function testspstat(isp,status)
! return value of species status bit
   implicit none
   integer isp,status
```

8.13.5 Get and test status for phase

Get and test phase status are almost identical but the get routine returns some additional informantion. Note that there are also phase bits that determine the model used.

```
 integer function get_phase_status(iph,ics,text,ip,val,ceq)
! return phase status as text and amount formula units in val
! for entered and fix phases also phase amounts.
! OLD Function value: 1=entered, 2=fix, 3=dormant, 4=suspended, 5=hidden
   implicit none
   character text*(*)
   integer iph,ics,ip
   TYPE(gtp_equilibrium_data), pointer :: ceq
   double precision val
 integer function test_phase_status(iph,ics,val,ceq)
! Almost same as get_..., returns phase status as function value but no text
! old: 1=entered, 2=fix, 3=dormant, 4=suspended, 5=hidden
! nystat:-4 hidden, -3 suspended, -2 dormant, -1,0,1 entered, 2 fix
! this is different from in change_phase .... one has to make up one's mind
   implicit none
   TYPE(gtp_equilibrium_data), pointer :: ceq
   integer iph,ics
   double precision val
```

8.13.6 Set/unset/test phase model bit

This sets bits indicating if the phase is ideal, has fcc ordering etc. It does not test if ENTERED/FIXED/DORMANT or SUSPENDED.

```
 subroutine set_phase_status_bit(lokph,bit)
! set the status bit "bit" in status1, cannot be done outside this module
! as the phlista is private
! These bits do not depend on the composition set
   implicit none
   integer lokph,bit
 subroutine clear_phase_status_bit(lokph,bit)
! clear the status bit "bit" in status1, cannot be done outside this module
! as the phlista is private
   implicit none
   integer lokph,bit
 logical function test_phase_status_bit(iph,ibit)
! return TRUE is status bit ibit for  phase iph, is set
! because phlista is private.  Needed to test for gas, ideal etc,
! DOES NOT TEST STATUS like entered/fixed/dormant/suspended
   implicit none
```

```
   integer iph,ibit
```

8.13.7 Change status for phase

Several subroutines to change the status bits for one or more phases. Note NYSTAT values are different from this in GET and TEST above. Same values should be used.

```
 subroutine change_many_phase_status(phnames,nystat,val,ceq)
! change the status of many phases.
! nystat:-4 hidden, -3 suspended, -2 dormant, -1,0,1 entered, 2 fix
! phnames is a list of phase names or *S (all suspeded) *D (all dormant) or
! *E (all entered (stable, unknown, unstable), *U all unstable
! If just * then change_phase_status is called directly
! It calls change_phase_status for each phase
   implicit none
   character phnames*(*)
   integer nystat
   double precision val
   TYPE(gtp_equilibrium_data), pointer :: ceq
 subroutine change_phtup_status(phtupx,nystat,val,ceq)
! change the status of a phase tuple. Also used when setting phase fix etc.
! nystat:-4 hidden, -3 suspended, -2 dormant, -1,0,1 entered, 2 fix
! qph can be -1 meaning all or a specifix phase index. ics compset
!
   implicit none
   integer phtupx,nystat
   double precision val
   TYPE(gtp_equilibrium_data), pointer :: ceq
 subroutine change_phase_status(qph,ics,nystat,val,ceq)
! change the status of a phase. Also used when setting phase fix etc.
! old: 0=entered, 1=suspended, 2=dormant, 3=fix, 4=hidden,5=not hidden
! nystat:-4 hidden, -3 suspended, -2 dormant, -1,0,1 entered, 2 fix
! qph can be -1 meaning all or a specifix phase index. ics compset
!
   implicit none
   integer qph,ics,nystat
   double precision val
   TYPE(gtp_equilibrium_data), pointer :: ceq
 subroutine mark_stable_phase(iph,ics,ceq)
! change the status of a phase. Does not change fix status
! called from meq_sameset to indicate stable phases (nystat=1)
! nystat:-4 hidden, -3 suspended, -2 dormant, -1,0,1 entered, 2 fix
!
   implicit none
   integer iph,ics
```

```
   TYPE(gtp_equilibrium_data), pointer :: ceq
```

8.13.8 Set unit for energy etc.

This is not yet implemented.

```
 subroutine set_unit(property,unit)
! set the unit for a property, like K, F or C for temperature
! >>>> unfinished
   implicit none
   character*(*) property,unit
```

8.13.9 Save results for a phase

This is not yet implemented.

```
 subroutine save_results(lut,iph,ics,long)
! write calculated results for a phase for later use in POST
   implicit none
   integer lut,iph,ics,long
```

8.13.10 Set reference state for constituent

This is not yet implemented.

```
 subroutine set_constituent_reference_state(iph,icon,asum)
! determine the end member to calculate as reference state for this constituent
! Used when giving a chemical potential for a constituent like MU(GAS,H2O)
   implicit none
   integer iph,icon
   double precision asum
```

8.13.11 Calculate conversion matrix for new components

```
 subroutine elements2components(nspel,stoi,ncmp,cmpstoi,ceq)
! converts a stoichiometry array for a species from elements to components
   implicit none
   integer nspel,ncmp
   double precision stoi(*),cmpstoi(*)
   type(gtp_equilibrium_data), pointer :: ceq
```

8.14 Internal stuff

The subroutines in this section are internal and not to be used by calls from outside GTP.

8.14.1 Indicate the type of an experiment

An experiment can specify an equal value, =, a larger than limit, > or a lesser than limit, <. This return the type.

```
 subroutine termterm(string,ich,kpos,value)
! search for first occurance of + - = > or <
! if + or - then also extract possible value after sign
   implicit none
   character string*(*)
   integer kpos,ich
   double precision value
```

8.14.2 Alphabetical ordering

The elements, species and phases are stored in the arrays ELLISTA, SPLISTA and PHLISTA in the order they are entered. They are sorted in alphabetical order in the array ELEMENTS, SPECIES and PHASES. These routines are used to maintain the alphabetical order.

```
 subroutine alphaelorder
! arrange new element in alphabetical order
! also make alphaindex give alphabetical order
   implicit none
 subroutine alphasporder
! arrange new species in alphabetical order
! also make alphaindex give alphabetical order
   implicit none
 subroutine alphaphorder(tuple)
! arrange last added phase in alphabetical order
! also make alphaindex give alphabetical order
! phletter G and L and I have priority
! tuple is returned as position in phase tuple
   implicit none
   integer tuple
```

8.14.3 Check alphaindex

Just for testing.

```
 subroutine check_alphaindex
```

```
! just for debugging, check that ellist(i)%alphaindex etc is  correct
   implicit none
```

8.14.4 Creates a list of constituents of a phase

Not much to discuss.

```
 subroutine create_constitlist(constitlist,nc,klist)
! creates a constituent list ...
   implicit none
   integer, dimension(*) :: klist
   integer, dimension(:), allocatable :: constitlist
   integer nc
```

8.14.5 Creates a new parrecord for a phase

Not much to discuss.

```
 subroutine create_parrecords(lokph,lokcs,nsl,nc,nprop,iva,ceq)
! fractions and results arrays for a phase for parallel calculations
! location is returned in lokcs
! nsl is sublattices, nc number of constituents, nprop max number if propert,
! iva is an array which is set as constituent status word (to indicate VA)
! ceq is always firsteq ???
!
! BEWARE not adopted for threads
!
! >>> changed all firsteq below to ceq????
!
   implicit none
   TYPE(gtp_equilibrium_data), pointer :: ceq
   integer, dimension(*) :: iva
   integer lokph,lokcs, nsl, nc, nprop
```

8.14.6 Create interaction record

Finds the correct place to add an interaction parameter. It should always be linked from the first possible endmember (with the alphabetically first constituents) and then in alphabetical order of the interaction elements.

```
 subroutine create_interaction(intrec,mint,lint,intperm,intlinks)
! creates a parameter interaction record
! with permutations if intperm(1)>0
```

```
   implicit none
   type(gtp_interaction), pointer :: intrec
   integer, dimension(2,*) :: lint,intlinks
   integer, dimension(*) :: intperm
   integer mint
```

8.14.7 Create endmember record

Creates a record for an endmember. Sometimes endmembers have no property records if there are interaction parameters must be linked from this endmember.

```
 subroutine create_endmember(lokph,newem,noperm,nsl,endm,elinks)
! create endmember record with nsl sublattices with endm as constituents
! noperm is number of permutations
! endm is the basic endmember
! elinks are the links to constituents for all permutations
   implicit none
   integer endm(*)
   type(gtp_endmember), pointer :: newem
   integer, dimension(nsl,noperm) ::elinks
   integer lokph,noperm,nsl
```

8.14.8 Create property record

All parameter values for an endmember or interaction record are stored in property records. An endmember or interaction parameter may have several property records linked in a list.

```
 subroutine create_proprec(proprec,proptype,degree,lfun,refx)
! reservs a property record from free list and insert data
   implicit none
   TYPE(gtp_property), pointer :: proprec
   integer proptype,degree,lfun
   character refx*(*)
```

8.14.9 Extend property record

An interaction record can have a degree and each degree a function. When entering a function for a higher degree than in the current property record it must be extended.

```
 subroutine extend_proprec(current,degree,lfun)
! extends a property record  and insert new data
   implicit none
   integer degree,lfun
   type(gtp_property), pointer :: current
```

8.14.10 Create a new phase_varres record

The phase_varres record belong to the dynamic dataset in the equilibrium record. When creating new equilibrium records or composition sets new phase_varres records are needed.

```
 subroutine new_phase_varres_record(iph,phvar,ceq)
! this subroutine returnes a copy of the phase variable structure for iph
! >>>>>>>>>>>>>
! this subroutine is probably redundant since the structure
! gtp_equilibrium_data was introduced.  Each parallel tread should have
! its own gtp_equilibrium_data record.
! >>>>>>>>>>>>>>>>>>>>>>>>>>
! The programmer can enter fraction in this structure and use it in calls
! to parcalcg should be suitable for parallel processing (NOT TESTED)
! when the same phase is calculated in several threads (like when separate
! threads calculate different lines suring mapping)
   implicit none
! >>>> unfinished
! >>>> for calculation of the same phase in separate threads
   integer iph
   TYPE(gtp_equilibrium_data) :: ceq
   TYPE(gtp_phase_varres) :: phvar
```

8.14.11 Add a disordered fraction set record

A phase with sublattices for long range ordering may have a fraction set representing the disordered phase. That can be for an fcc phase where the disordered fractions represent the disordered state or for a sigma phase where the disordered fractions represent some kind of hypothetical state. A disordered fraction set can have its own set of parameters,

```
 subroutine new_disordered_phase_variable_record(lokdis,phvar,phdis,ceq)
! Does this really work????
! creates a copy of the disordered phase variable record lokdis
! and set links from ordered phvar
!   ?????????????? does this work ??????????  is it necessary ????
! can one just make an assignment ????
   implicit none
   TYPE(gtp_equilibrium_data) :: ceq
   TYPE(gtp_phase_varres) :: phvar
   TYPE(gtp_phase_varres), target :: phdis
   integer lokdis
```

8.14.12 Add a fraction set record

```
 subroutine add_fraction_set(iph,id,ndl,totdis)
```

```
! add a new set of fractions to a phase, usually to describe a disordered state
! like the "partitioning" in old TC
!
! BEWARE this is only done for firsteq, illegal when having more equilibria
!
! id is a letter used as suffix to identify the parameters of this set
! ndl is the last original sublattice included in the (first) disordered set
! ndl can be 1 meaning sublattice 2..nsl are disordered, or nsl meaning all are
!     disordered
! totdis=0 if phase never disorder totally (like sigma)
!
! For a phase like (Al,Fe,Ni)3(Al,Fe,Ni)1(C,Va)4 to add (Al,Fe,Ni)4(C,Va)4
! icon=1 2 3 1 2 3 4 5 with ndl=2
! For a phase like (Fe,Ni)10(Cr,Mo)4(Cr,Fe,Mo,Ni)16 then
! icon=2 4 1 3 1 2 3 4 with ndl=3
! This subroutine will create the necessary data to calculate the
! disordered fraction set from the site fractions.
!
! IMPORTANT (done): for each composition set this must be repeated
! if new composition sets are created it must be repeated for these
!
! IMPORTANT (not done): order the constituents alphabetically in each disorderd
! sublattice otherwise it will not be possible to enter parameters correctly
!
   implicit none
   integer iph,ndl,totdis
   character id*1
```

8.14.13 Copy record for fraction sets

```
 subroutine copy_fracset_record(lokcs,disrec,ceq)
! attempt to create a new disordered record  ??? this can probably be done
! with just one statement .. but as it works I am not changing right now
   implicit none
   TYPE(gtp_equilibrium_data) :: ceq
   TYPE(gtp_fraction_set) :: disrec
   integer lokcs
```

8.14.14 Implicit suspend and restore

If an element is suspended some species may have to be suspended too and if a species is suspended some phases may have to be suspended. This routine does that.

```
 subroutine suspend_species_implicitly(ceq)
! loop through all entered species and suspend those with an element suspended
```

```
   implicit none
   TYPE(gtp_equilibrium_data), pointer :: ceq
 subroutine suspend_phases_implicitly(ceq)
! loop through all entered phases and suspend constituents and
! SUSPEND phases with all constituents in a sublattice suspended
!   dimension lokcs(9)
   implicit none
   TYPE(gtp_equilibrium_data) :: ceq
 subroutine restore_species_implicitly_suspended
! loop through all implicitly suspended species and restore those with
! all elements enteded
   implicit none
 subroutine restore_phases_implicitly_suspended
! loop through all implicitly suspended phases and restore those with
! at least one constituent entered in each sublattice
   implicit none
```

8.14.15 Add to reference phase

There is a reference phase that should have parameters for each element it its stable state at all temperatures and 1 bar. For each element entered this subroutine adds it to the reference phase. This phase can never be used in calculations. It represent different phases for each element, gas for H, bcc for Cr etc.

```
 subroutine add_to_reference_phase(loksp)
! add this element to the reference phase
! loksp: species index of new element
   implicit none
   integer loksp
```

8.15 Additions

Creating, handling and calculations of additions to the Gibbs energy. This is a section that will probably be extended with several new subroutines for different kinds of additions.

```
 subroutine addition_selector(addrec,moded,phres,lokph,mc,ceq)
! called when finding an addtion record while calculating G for a phase
! addrec is addition record
! moded is 0, 1 or 2 if no, first or 2nd order derivatives should be calculated
! phres is ?
! lokph is phase location
! mc is number of constitution fractions
! ceq is current equilibrium record
   implicit none
```

```
   type(gtp_phase_add), pointer :: addrec
   integer moded,lokph,mc
   TYPE(gtp_phase_varres), pointer :: phres
   type(gtp_equilibrium_data), pointer :: ceq
```

8.15.1 Generic subroutine to add an addition

Not well structured here.

```
 subroutine add_addrecord(iph,addtyp)
! generic subroutine to add an addition typ addtyp (Except Inden)
   implicit none
   integer iph,addtyp
```

8.15.2 Utility routine for addition

Searches for the composition dependent properties for a specific addition.

```
 subroutine need_propertyid(id,typty)
! get the index of the property needed
   implicit none
   integer typty
   character*4 id
```

8.15.3 Enter and calculate Inden magnetic model

The first two subroutines create the ferromagnetic addition due to Inden model and store all necessary data inside this. The last subroutine is called when calculating the Gibbs energy for a phase if there is a magnetic addition linked to the phase. It must calculate the contribution to G and all first and second derivatives of G.

In the call the pointer to the phase_varres record is provided where current values of G and derivatives can be found. Values of the ferromagnetic temperature and its derivatives with respect to constitution is also stored there. The chain rule for derivatives must be applied.

```
 subroutine add_magrec_inden(lokph,addtyp,aff)
! adds a magnetic record to lokph
! lokph is phase location
! addtyp should be 1 of Inden model
! aff is antiferromagnic factor, -1 for bcc and -3 for fcc and hcp
   implicit none
   integer lokph,addtyp,aff
 subroutine create_magrec_inden(addrec,aff)
! enters the magnetic model
```

```
   implicit none
   type(gtp_phase_add), pointer :: addrec
   integer aff
 subroutine calc_magnetic_inden(moded,phres,lokadd,lokph,mc,ceq)
! calculates Indens magnetic contribution
! NOTE: values for function not saved, should be done to save time.
! Gmagn = RT*f(T/Tc)*ln(beta+1)
! moded: integer, 0=only G, S, Cp; 1=G and dG/dy; 2=Gm dG/dy and d2G/dy2
! phres: pointer, to phase\_varres record
! lokadd: pointer, to addition record
! lokph: integer, phase record
! mc: integer, number of constituents
! ceq: pointer, to gtp_equilibrium_data
   implicit none
   integer moded,lokph,mc
   TYPE(gtp_phase_varres) :: phres
   TYPE(gtp_phase_add), pointer :: lokadd
   TYPE(gtp_equilibrium_data) :: ceq
```

8.15.4 Create new magnetic model

Wei Xiagong has proposed a simplified magnetic model. It is not yet implemented.

```
 subroutine create_weimagnetic(addrec,bcc)
! adds a wei type magnetic record, we must separate fcc and bcc but no aff!!
! copied from Inden magnetic model
! The difference is that it uses TCA for Curie temperature and TNA for Neel
! and individual Bohr magneton numbers
   implicit none
   logical bcc
   type(gtp_phase_add), pointer :: addrec
 subroutine calc_weimagnetic(moded,phres,lokadd,lokph,mc,ceq)
! calculates Wei-Indens magnetic contribution
!
! NOTE this is just copied from Inden subroutine, must be changed
!
! Gmagn = RT*f(T/Tc)*ln(beta+1)
! moded: integer, 0=only G, S, Cp; 1=G and dG/dy; 2=Gm dG/dy and d2G/dy2
! phres: pointer, to phase\_varres record
! lokadd: pointer, to addition record
! lokph: integer, phase record
! mc: integer, number of constituents
! ceq: pointer, to gtp_equilibrium_data
   implicit none
   integer moded,lokph,mc
```

```
! phres points to result record with gval etc for this phase
   TYPE(gtp_phase_varres) :: phres
   TYPE(gtp_phase_add), pointer :: lokadd
   TYPE(gtp_equilibrium_data) :: ceq
```

8.15.5 Calculate and calculate elastic contribution

This creates and calculates the elastic record.

```
 subroutine create_elastic_model_a(newadd)
! addition record to calculate the elastic energy contribution
   implicit none
   type(gtp_phase_add), pointer :: newadd
 subroutine calc_elastica(moded,phres,addrec,lokph,mc,ceq)
! calculates elastic contribution and adds to G and derivatives
   implicit none
   integer moded,lokph,mc
   type(gtp_phase_varres), pointer :: phres
   type(gtp_phase_add), pointer :: addrec
   type(gtp_equilibrium_data), pointer :: ceq
 subroutine set_lattice_parameters(iph,ics,xxx,ceq)
! temporary way to set current lattice parameters for use with elastic model a
   implicit none
   integer iph,ics
   double precision, dimension(3,3) :: xxx
   type(gtp_equilibrium_data) :: ceq
```

8.15.6 Heat capacity model for Einstein solids

Unfinished

```
 subroutine create_einsteincp(newadd)
   implicit none
   type(gtp_phase_add), pointer :: newadd
 subroutine calc_einsteincp(moded,phres,addrec,lokph,mc,ceq)
! Calculate the contibution due to Einste Cp model for low T
! moded 0, 1 or 2
! phres all results
! addrec pointer to addition record
! lokph phase record
! mc number of variable fractions
! ceq equilibrum record
!
! G = 3*R*T*ln( 1 - exp( THET/T ) )
```

```
! This is easier to handle inside the calc routine without TPFUN
!
   implicit none
   integer moded,lokph,mc
   type(gtp_phase_varres), pointer :: phres
   type(gtp_phase_add), pointer :: addrec
   type(gtp_equilibrium_data), pointer :: ceq
```

8.15.7 Glas addition

Unfinished

```
 subroutine create_glas_transition_modela(newadd)
! not implemented
   implicit none
   type(gtp_phase_add), pointer :: newadd
```

8.15.8 Debye heat capacity model

These subroutines are called to create and calculate the Gibbs energy contribution for a phase if there is a Debye model addition linked to the phase. It must calculate the contribution to G and all first and second derivatives of G. Careful study of the Inden magnetic model record is recommended.

In the call the pointer to the phase_varres record is provided where current values of G and derivatives can be found. Values of the Debye temperature and its derivatives with respect to constitution is also stored there. The chain rule for derivatives must be applied.

Unfinished.

```
 subroutine create_debyecp(addrec)
! enters a record for the debye model
   implicit none
   type(gtp_phase_add), pointer :: addrec
 subroutine calc_debyecp(moded,phres,lokadd,lokph,mc,ceq)
! calculates Mauro Debye contribution
! NOTE: values for function not saved, should be done to save calculation time.
! moded: integer, 0=only G, S, Cp; 1=G and dG/dy; 2=Gm dG/dy and d2G/dy2
! phres: pointer, to phase\_varres record
! lokadd: pointer, to addition record
! lokph: integer, phase record
! mc: integer, number of constituents
! ceq: pointer, to gtp_equilibrium_data
   implicit none
   integer moded,lokph,mc
```

```
   TYPE(gtp_equilibrium_data) :: ceq
   TYPE(gtp_phase_add), pointer :: lokadd
   TYPE(gtp_phase_varres) :: phres
```

8.15.9 List additions

When listing data for a phase with addition the relevant information must be listed also for additions.

```
 subroutine list_addition(unit,ch1,phname,ftyp,lokadd)
! list description of an addition for a phase on unit
   implicit none
   integer unit,ftyp
   character ch1*1,phname*(*)
   TYPE(gtp_phase_add), pointer :: lokadd
```

8.16 Calculation

There are many subroutines involved in calculating the Gibbs energy for a system and to retrieve values afterwards. Some are explained in connection with what they calculate, for example the magnetic contribution in 8.15.3.

8.16.1 Calculate for one phase

This subroutine calculates the Gibbs energy and all first and second derivatives with respect to T, P and constituents for the specified phase and composition set using the current values of T, P and constitution of the phase (set by set_constitution, see 8.6.1). It also calculates all other properties stored in the property records.

It is possible to calculate only G my setting moded=0, only G and first derivatives if moded=1 and also second derivatives with moded=2. This routine calls calcg_internal to do the calculations after some checks.

```
 subroutine calcg(iph,ics,moded,lokres,ceq)
! calculates G for phase iph and composition set ics in equilibrium ceq
! checks first that phase and composition set exists
! Data taken and stored in equilibrium record ceq
! lokres is set to the phase_varres record with all fractions and results
! moded is 0, 1 or 2 depending on calculating no, first or 2nd derivarives
   implicit none
   TYPE(gtp_equilibrium_data), pointer :: ceq
   integer iph,ics,moded,lokres
```

8.16.2 Model independent routine for one phase calculation

This is the central subroutine to calculate G and derivatives for all kinds of phases. At present only the CEF and ionic liquid model is implemented. It calls many other calculating subroutines, some are described in connection with the property they calculate, like magnetic contribution.

```
 subroutine calcg_internal(lokph,moded,cps,ceq)
! Central calculating routine calculating G and everyting else for a phase
! ceq is the equilibrium record, cps is the phase_varres record for lokph
! moded is type of calculation, 0=only G, 1 G and first derivatives
!    2=G and all second derivatives
! Can also handle the ionic liquid model now ....
   implicit none
   integer lokph,moded
   TYPE(gtp_equilibrium_data), pointer :: ceq
   TYPE(gtp_phase_varres), target :: cps
```

8.16.3 A utility routine

This is used when a phase has permutations, see 5.6.1

```
 subroutine setendmemarr(lokph,ceq)
! stores the pointers to all ordered and disordered endmemners in arrays
   implicit none
   integer lokph
   TYPE(gtp_equilibrium_data), pointer :: ceq
```

8.16.4 Calculate and list results for one phase

This is mainly a debugging routine that calculates and lists for a specific phase the Gibbs energy and all first and second derivatives by calling calcg using the current values of T, P and constitution. It does not iterate and can thus not calculate an equilibrium.

```
 subroutine tabder(iph,ics,ceq)
! tabulate derivatives of phase iph with current constitution and T and P
   implicit none
   integer iph,ics
   TYPE(gtp_equilibrium_data), pointer :: ceq
```

8.16.5 Calculate an interaction parameter

This is called by calcg_internal to calculate the value of an interaction parameter and its derivatives and add this to all property arrays.

```
 subroutine cgint(lokph,lokpty,moded,vals,dvals,d2vals,gz,ceq)
! calculates an excess parameter that can be composition dependent
! gz%yfrem are the site fractions in the end member record
! gz%yfrint are the site fractions in the interaction record(s)
! lokpty is the property index, lokph is the phase record
! moded=0 means only G, =1 G and dG/dy, =2 all
   implicit none
   integer moded,lokph
   TYPE(gtp_property), pointer :: lokpty
   TYPE(gtp_parcalc) :: gz
   double precision vals(6),dvals(3,gz%nofc)
   TYPE(gtp_equilibrium_data) :: ceq
```

8.16.6 Calculate ideal configurational entropy

This calculates the ideal configurational entropy summed over all sublattices.

```
 subroutine config_entropy(moded,nsl,nkl,phvar,tval)
! calculates configurational entropy/R for phase lokph
   implicit none
   integer moded,nsl
   integer, dimension(nsl) :: nkl
   TYPE(gtp_phase_varres), pointer :: phvar
```

8.16.7 Calculate ionic liquid configurational entropy

The ionic liquid model assumes ideal mixing on each sublattice but the site ratios are not constant.

```
 subroutine config_entropy_i2sl(moded,nsl,nkl,phvar,i2slx,tval)
! calculates configurational entropy/R for ionic liquid model
! Always 2 sublattices, the sites depend on composition
! P = \sum_j (-v_j) y_j + Q y_Va
! Q = \sum_i v_i y_i
! where v is the charge on the ions. P and Q calculated by set_constitution
   implicit none
   integer moded,nsl,i2slx(2)
   integer, dimension(nsl) :: nkl
   TYPE(gtp_phase_varres), pointer :: phvar
```

8.16.8 Push/pop constituent fraction product on stack

These subroutines are used to push/pop current values of the product of constituent fractions and its derivatives before calculating an interaction parameter.

```
 subroutine push_pyval(pystack,intrec,pmq,pyq,dpyq,d2pyq,moded,iz)
! push data when entering an interaction record
   implicit none
   integer pmq,moded,iz
   double precision pyq,dpyq(iz),d2pyq(iz*(iz+1)/2)
   type(gtp_pystack), pointer :: pystack
   type(gtp_interaction), pointer :: intrec
 subroutine pop_pyval(pystack,intrec,pmq,pyq,dpyq,d2pyq,moded,iz)
! pop data when entering an interaction record
   implicit none
   integer iz,pmq,moded
   double precision pyq,dpyq(iz),d2pyq(iz*(iz+1)/2)
   type(gtp_pystack), pointer :: pystack
   type(gtp_interaction), pointer :: intrec
```

8.16.9 Calculate disordered fractions from constituent fractions

This is used when there are several fraction sets of a phase. The values of the second fraction set (also called the disordered fraction set) is calculated by this subroutine. These disordered fractions can be used to calculate a "disordered" part of the Gibbs energy with its own set of parameters.

```
 subroutine calc_disfrac(lokph,lokcs,ceq)
! calculate and set disordered set of fractions from sitefractions
! The first derivatives are dxidyj.  There are no second derivatives
!   TYPE(gtp_fraction_set), pointer :: disrec
   implicit none
   integer lokph,lokcs
   TYPE(gtp_equilibrium_data), pointer :: ceq
```

8.16.10 Disorder constituent fractions

The Gibbs energy for the "partitioned" phases, like FCC and BCC which can have order/disorder transformations, see 2.5, the "ordered part" is calculated twice, once with the original constituent fractions and once with these set equal to their disordered value. The reason for this is that the "disordered part" should be complete i.e. include also the disordered part of the "ordered part" as the disordered partitions.

This is now an optional way to handle partioning, the recommended way is to simply add the Gibbs energy is simply added from the two fraction sets. This subroutine sets the fractions to their disordered values.

```
 subroutine disordery(phvar,ceq)
! sets the ordered site fractions in FCC and other order/disordered phases
! equal to their disordered value in order to calculate and subtract this part
```

```
! phvar is index to phase_varres for ordered fractions
   implicit none
   TYPE(gtp_phase_varres), pointer :: phvar
   TYPE(gtp_equilibrium_data) :: ceq
```

8.16.11 Set driving force for a phase explicitly

Another failed attempt to handle convergence problems.

```
 subroutine set_driving_force(iph,ics,dgm,ceq)
! set the driving force of a phase explicitly
   implicit none
   type(gtp_equilibrium_data), pointer :: ceq
   integer iph,ics
   double precision dgm
```

8.16.12 Extract mass balance conditions

This is used in global grid minimization to extract the set of mass balance conditions. If the current conditions are not all mass balance there is an error return, otherwise the conditions of T and P, the total number of moles and the mole fractions of all components are returned.

```
 subroutine extract_massbalcond(tpval,xknown,antot,ceq)
! extract T, P,  mol fractions of all components and total number of moles
! for use when minimizing G for a closed system.  Probably redundant
   implicit none
   double precision, dimension(*) :: tpval,xknown
   double precision antot
   TYPE(gtp_equilibrium_data), pointer :: ceq
```

8.16.13 Saving and restoring a phase constitution

These are used during step and map to help handling convergence problems

```
 subroutine save_constitutions(ceq,copyofconst)
! copy the current phase amounts and constituitions to be restored
! if calculations fails during step/map
! DANGEROUS IF NEW COMPOSITION SETS CREATED
   implicit none
   TYPE(gtp_equilibrium_data), pointer :: ceq
   double precision, allocatable, dimension(:) :: copyofconst
 subroutine restore_constitutions(ceq,copyofconst)
! restore the phase amounts and constituitions from copyofconst
```

```
! if calculations fails during step/map
! DANGEROUS IF NEW COMPOSITION SETS CREATED
   implicit none
   TYPE(gtp_equilibrium_data), pointer :: ceq
   double precision copyofconst(*)
```

8.17 Grid minimizer

When there is only mass balance conditions it is possible to use a global minimization technique to find start points for a more standard Newton-Raphson technique. The latter has the disadvantage that it can only find local minimas.

Tn the global minimization all solution phases are divided into a grid with fixed compositions and the Gibbs energy for each gridpoint is calculated. These gridpoints are then searched until one finds a set that encloses the given overall composition which represent the minimal Gibbs energy. It is thus necessary to know the overall composition.

The phases the gridpoints in the solution are then identified and possibly one can merge gridpoints in the same phase unless they are separated by a miscibility gap.

Some care should be taken with ordered phases as they have several identical sublattices and one thus can reduce significantly the number of gridpoints generated if this is taken into account.

8.17.1 Global Gridminimizer

This is the main grid minimizing subroutine. It should be modified to be callable also after an equilibrium calculation to check if there are any phases below the calculated G hypersurface.

```
 subroutine global_gridmin(what,tp,xknown,nvsph,iphl,icsl,aphl,nyphl,cmu,ceq)
!
! finds a set of phases that is a global start point for an equilibrium
! calculation at T and P values in tp, total amount of atoms in totan
! and known mole fraction in xknown
! It is intentional that this routine is independent of current conditions
! It returns: nvsph stable phases, list of phases in iphl, amounts in aphl,
! nyphl(i) is redundant, cmu are element chemical potentials of solution
! WHAT determine what to do with the results, 0=just return solution,
! 1=enter stable set and constitution of all phases in gtp datastructure
! and create composition sets if necessary (and allowed)
! what=-1 will check if any gridpoint below current calculated equilibrium
   implicit none
! nyphl(j) is the start position of the constitiuent fractions of phase j in
   integer, dimension(*) :: iphl,nyphl,icsl
   integer what,nvsph
   TYPE(gtp_equilibrium_data), pointer :: ceq
```

```
   double precision, dimension(2) :: tp
! cmu(1..nrel) is the chemical potentials of the solution
   double precision, dimension(*) :: xknown,aphl,cmu
```

8.17.2 Generate grid

There are several different grid generators depending on the type of phase for example ionic or with order/disorder transformation. It is also a possibility to generate a denser grid.

```
 subroutine generate_grid(mode,iph,ngg,nrel,xarr,garr,ny,yarr,gmax,ceq)
! Different action depending of the value of mode,
! for mode<0:
!    return the number of gridpoints that will be generated for phase iph in ngg
! for mode=0:
!    return garr(i) gibbs energy and xarr(1,i) the compositions of gridpoint i
! for mode>0:
!    return site fractions of gridpoint mode in yarr, number of fractions in ny
!    iph is phase number, ngg is number of gridpoints, nrel number of elements,
! if mode=0:
!    return xarr mole fractions of gridpoints, garr Gibbs energy of gridpoints,
!    ngg is dimension of garr, gmax maximum G (start value for chem.pot)
! if mode>0:
!   "mode" is a gridpoint of this phase in solution, return number of
!   constituent fractions in ny and fractions in yarr for this gridpoint
! The current constitution is restored at the end of the subroutine
   implicit none
   TYPE(gtp_equilibrium_data), pointer :: ceq
   integer mode,iph,ngg,nrel,ny
   real xarr(nrel,*),garr(*)
   double precision yarr(*),gmax
 subroutine generate_dense_grid(mode,iph,ngg,nrel,xarr,garr,ny,yarr,gmax,ceq)
! generates more gridpoints than default generate_grid
! Different action depending of the value of mode,
! for mode<0:
!    return the number of gridpoints that will be generated for phase iph in ngg
! for mode=0:
!    return garr(i) gibbs energy and xarr(1,i) the compositions of gridpoint i
! for mode>0:
!    return site fractions of gridpoint mode in yarr, number of fractions in ny
!    iph is phase number, ngg is number of gridpoints, nrel number of elements,
! if mode=0:
!    return xarr mole fractions of gridpoints, garr Gibbs energy of gridpoints,
!    ngg is dimension of garr
! if mode>0:
!   "mode" is a gridpoint of this phase in solution, return number of
```

```
!   constituent fractions in ny and fractions in yarr for this gridpoint
! The current constitution is restored at the end of the subroutine
   implicit none
   TYPE(gtp_equilibrium_data), pointer :: ceq
   integer mode,iph,ngg,nrel,ny
   real xarr(nrel,*),garr(*)
   double precision yarr(*),gmax
 subroutine generate_fccord_grid(mode,iph,ngg,nrel,xarr,garr,ny,yarr,gmax,ceq)
! This generates grid for a phase with 4 sublattice fcc/hcp ordering
! mode<0 just number of gridpoints in ngg, needed for allocations
! mode=0 calculate mole fraction and G for all gridpoints
! mode>0 return constitution for gridpoint mode in yarr
   implicit none
   integer mode,iph,ngg,nrel,ny
   real xarr(nrel,*),garr(*)
   double precision yarr(*),gmax
   type(gtp_equilibrium_data), pointer :: ceq
 subroutine generate_charged_grid(mode,iph,ngg,nrel,xarr,garr,ny,yarr,gmax,ceq)
! This generates grid for a phase with charged constituents
! mode<0 just number of gridpoints in ngg, needed for allocations
! mode=0 calculate mole fraction and G for all gridpoints
! mode>0 return constitution for gridpoint mode in yarr
   implicit none
   integer mode,iph,ngg,nrel,ny
   real xarr(nrel,*),garr(*)
   double precision yarr(*),gmax
   type(gtp_equilibrium_data), pointer :: ceq
```

8.17.3 Calculate gridpoint

The Gibbs energy for each gridpoint is calculated as well as its composition.

```
 subroutine calc_gridpoint(iph,yfra,nrel,xarr,gval,ceq)
! called by global minimization routine
! Not adopted to charged crystalline phases as gridpoints have net charge
! but charged gripoints have high energy, better to look for neutral ones ...
   implicit none
   real xarr(*),gval
   integer iph,nrel
   double precision yfra(*)
   TYPE(gtp_equilibrium_data), pointer :: ceq
```

8.17.4 Calculate gridpoints for some special phase models

The ordered fcc, bcc and hcp with 4 sublattice model can have significantly reduced number of gridpoints. This is not yet implemented.

The same when a phase have charged constituents with many anemembers that have net charge. This is implemented but not optimized.

```
 subroutine generate_fccord_grid(mode,iph,ngg,nrel,xarr,garr,ny,yarr,ceq)
! This generates grid for a phase with 4 sublattice fcc/hcp ordering
! mode<0 just number of gridpoints in ngg, needed for allocations
! mode=0 calculate mole fraction and G for all gridpoints
! mode>0 return constitution for gridpoint mode in yarr
   implicit none
   integer mode,iph,ngg,nrel,ny
   real xarr(nrel,*),garr(*)
   double precision yarr(*)
   type(gtp_equilibrium_data), pointer :: ceq
 subroutine generate_charged_grid(mode,iph,ngg,nrel,xarr,garr,ny,yarr,ceq)
! This generates grid for a phase with charged constituents
! mode<0 just number of gridpoints in ngg, needed for allocations
! mode=0 calculate mole fraction and G for all gridpoints
! mode>0 return constitution for gridpoint mode in yarr
   implicit none
   integer mode,iph,ngg,nrel,ny
   real xarr(nrel,*),garr(*)
   double precision yarr(*)
   type(gtp_equilibrium_data), pointer :: ceq
```

8.17.5 Calculate endmember

This is used by the calc_gridpoint routine.

```
 subroutine calcg_endmember(iph,endmember,gval,ceq)
! calculates G for one mole of real atoms for a single end member
! used for reference states. Restores current composition (but not G or deriv)
! endmember contains indices in the constituent array, not species index
! one for each sublattice
   implicit none
   integer iph
   double precision gval
   integer endmember(maxsubl)
   TYPE(gtp_equilibrium_data), pointer :: ceq
 subroutine calcg_endmember6(iph,endmember,gval,ceq)
! calculates G and all derivatevs wrt T and P for one mole of real atoms
```

```
! for a single end member, used for reference states.
! Restores current composition and G (but not deriv)
! endmember contains indices in the constituent array, not species index
! one for each sublattice
   implicit none
   integer iph
   double precision gval(6)
   integer endmember(maxsubl)
   TYPE(gtp_equilibrium_data), pointer :: ceq
```

8.17.6 Calculate minimum of grid

The search starts from the lowest G value for the pure elements. (This seems obvious but makes it impossible to have a grid for only fcc with carbon dissolved as fcc does not exist for pure carbon). The selected gridpoints always represent a hyperplane (number of dimensions equal to that of the number of components) of Gibbs energy over the whole composition range. Then the whole grid is searched for the gridpoint that has the most negative deviation from this hyperplane. Then one gridpoint in the existing hyperplane is replaced by this in such a way that the overall mass balance is inside the points forming the plane. Then a new search is made and a point is replaced until there are no points below the hyperplane. The points forming this plane represent the solution. There will always be as many points as components.

```
 subroutine find_gridmin(kp,nrel,xarr,garr,xknown,jgrid,phfrac,cmu,trace)
! there are kp gridpoints, nrel is number of components
! composition of each gridpoint in xarr, G in garr
! xknown is the known overall composition
! return the gridpoints of the solution in jgrid, the phase fraction in phfrac
! cmu are the final chemical potentials
   implicit none
   integer, parameter :: jerr=50
   integer kp,nrel
   integer, dimension(*) :: jgrid
   real xarr(nrel,*),garr(*)
   double precision xknown(*),phfrac(*),cmu(nrel)
   logical trace
```

8.17.7 Merge gridpoints in same phase

This subroutine tries to check if the number of gridpoints can be reduced by merging gridpoints in the same phase. Care must be taken that there can be miscibility gaps between points.

This subroutine may automatically create new composition sets if necessary (unless the user has set the appropriate bit to prevent that).

```
 subroutine merge_gridpoints(nv,iphl,aphl,nyphl,yphl,trace,nrel,xsol,cmu,ceq)
!
! BEWARE not adopted for parallel processing
!
! if the same phase has several gridpoints check if they are really separate
! (miscibility gaps) or if they can be murged.  Compare them two by two
! nv is the number of phases, iphl(i) is the index of phase i, aphl(i) is the
! amount of phase i, nyphl is the number of site fractions for phase i,
! and yphl is the site fractions packed together
!
   implicit none
   TYPE(gtp_equilibrium_data), pointer :: ceq
   integer nv,nrel
   integer, dimension(*) :: iphl,nyphl
   double precision, dimension(*) :: aphl,yphl,cmu
   logical trace
   real xsol(maxel,*)
```

8.17.8 Set constitution of metastable phases

Phases not part of the final solution will have their constitution set to a gridpoint that is closest to the final hyperplane.

```
 subroutine set_metastable_constitutions(ngg,nrel,kphl,ngrid,xarr,garr,&
      nr,iphl,cmu,ceq)
! this subroutine goes through all the metastable phases
! after a global minimization and sets the constituion to the most
! favourable one.  Later care should be taken that composition set 2
! and higher are not set identical or equal to the stable
! nrel  number of components
! ngg   number of gridpoints
! kphl  array with first points calculated for phase(i) in garr
! ngrid array with last  points calculated for phase(i) in garr
! garr  array with Gibbs energy/RT for each gridpoint
! xarr  matix with composition in all gridpoints
! nr    is the number of stable phases in the solution
! iphl  array with the phase numbers of the stable phases (not ordered)
! cmu   are the chemical potentials/RT of the solution
! ceq   equilibrium record
! called by global_gridmin
   implicit none
   integer ngg,nrel,nr
   integer, dimension(*) :: kphl,ngrid,iphl
   double precision, dimension(*) :: cmu
   real garr(*),xarr(nrel,*)
```

```
   type(gtp_equilibrium_data), pointer :: ceq
```

8.17.9 Check of equilibrium using grid minimizer

This subroutine can be used after an equilibrium calculation when the conditions does not allow the grid minimizer to be used (if T, P are not known or all conditions are not mass balance). It is not yet implemented.

```
 subroutine gridmin_check(nystph,kp,nrel,xarr,garr,xknown,ngrid,pph,&
      cmu,yphl,iphx,ceq)
! This subroutine checks if a calculated solution is correct by
! checking if there are any gridpoints below the surface defined
! by the chemical potentials cmu
! nystph return 0 or 10*(phase number)+compset number for new stable phase
! there are kp gridpoints, nrel is number of components
! composition of each gridpoint in xarr, G in garr
! xknown is the known overall composition
! ngrid is last  calculated gridpoint point for a phase jj
! pph is number of phases for which there is a grid
! iphx is phase numbers
! cmu are the final chemical potentials
! yphl is just needed as a dummy
! ceq is current equilibrium record
   implicit none
   integer kp,nrel,jp,ie,mode,pph,nystph
   double precision, parameter :: phfmin=1.0D-8
   real xarr(nrel,*),garr(*)
   double precision xknown(*)
   integer, dimension(*) :: ngrid,iphx
   double precision cmu(*),gsurf,gstable,gd,yphl(*),qq(5),rtn,gdmin
   TYPE(gtp_equilibrium_data), pointer :: ceq
```

8.18 Miscellaneous things

Things that does not fit anywhere else.

8.18.1 Phase record location

```
 integer function phvarlok(lokph)
! return index of the first phase_varres record for phase with location lokph
! needed for external routines as phlista is private
   implicit none
   integer lokph
```

8.18.2 Numbers an interaction tree for permutations

For permutations it turned out to be necessary to keep track of the individual interaction records to know the permutation. This subroutine indexes the interaction records for each endmember of a phase.

```
 subroutine palmtree(lokph)
! Initiates a numbering of all interaction trees of an endmember of a phase
   implicit none
   integer lokph
```

8.18.3 Check that certain things are allowed

To have a uniform check if something is allowed. One must not enter a phase before there are any elements for example.

```
 logical function allowenter(mode)
! Check if certain commands are allowed
! mode=1 means entering an element or species
! mode=2 means entering a phase
! mode=3 means entering an equilibrium
! returns TRUE if command can be executed
   implicit none
   integer mode
```

8.18.4 Check proper symbol

There are many symbols and names in the GTP package. In general a symbol must start with a letter A-Z. All symbols are also case insensitive, i.e. upper and lower case are treated as the same. Most symbols may contain digits as second and later character. Some symbols and names may contain special characters like "_" and others.

This subroutine checks this.

```
 logical function proper_symbol_name(name,typ)
! checks that name is a proper name for a symbol
! A proper name must start with a letter A-Z
! for typ=0 it must contain only letters, digits and underscore
! for typ=1 it may contain also +, - maybe ?
! It must not be equal to a state variable
   implicit none
   integer typ
   character name*(*)
```

8.18.5 The amount of a phase is set to a value

I am not sure when this is used or needed.

```
 subroutine set_phase_amounts(jph,ics,val,ceq)
! set the amount formula units of a phase. Called from user i/f
! iph can be -1 meaning all phases, all composition sets
   implicit none
   integer jph,ics
   double precision val
   TYPE(gtp_equilibrium_data), pointer :: ceq
```

8.18.6 Set the default constitution of a phase

I am not sure when this is used and if it is needed.

```
 subroutine set_default_constitution(iph,ics,ceq)
! the current constitution of (iph#ics) is set to its default constitution
!  (if any), otherwise a random value.  The amount of the phase not changed
   implicit none
   integer iph,ics
   TYPE(gtp_equilibrium_data), pointer :: ceq
```

8.18.7 Subroutine to prepare for an equilibrium calculation

Unfinished

```
 subroutine todo_before(mode,ceq)
! this could be called before an equilibrium calculation
! It should remove any phase amounts and clears CSSTABLE
! DUMMY
!
   implicit none
   TYPE(gtp_equilibrium_data), pointer :: ceq
   integer mode
```

8.18.8 Subroutine to clean up after an equilibrium calculation

This now checks if there are any composition sets with the AUTO bit set meaning that they have been created by the grid minimizer in this equilibrium calculation. If so it tries to move the stable phases to the lowest possible composition set, taking care that user defined composition sets with a given default constitution is honored, i.e a fcc carbide is not set as composition set 1 if the user has defined a metallic fcc phase with low carbon as the first.

It then removes unstable composition sets with the AUTO bit set and any stable composition sets with the AUTO bit set have this bit cleared.

```
 subroutine todo_after_found_equilibrium(mode,ceq)
! this is called after an equilibrium calculation
! It marks stable phase (set CSSTABLE and remove any CSAUTO)
! It removes redundant unstable composition sets created automatically
! (CSAUTO set).  It will also shift stable composition sets to loweest
! possible (it will take into account if there are default constituent
! fractions, CSDEFCON set).
! mode determine some of the actions
!
! >>>>>>>>>>> THIS IS DANGEROUS IN PARALLEL PROCESSING
! It should work in step and map as a composition set that once been stable
! will never be removed except if one does global minimization during the
! step and map. Then  metallic-FCC and MC-carbides may shift composition sets.
! Such shifts should be avoided by manual entering of comp.sets with
! default constitutions, but comparing a stable constitution with a
! default is not trivial ...
!
   implicit none
   TYPE(gtp_equilibrium_data), pointer :: ceq
   integer mode
```

8.18.9 Select composition set for stable phase

After an equilibrium calculation there may be automatically created composition sets by the gridminimizer that are not needed. These routines tries to remove unneeded sets and also shift the used ones to the lowest composition set. It tries to take into account the default constitutions for user defined composition sets for example an fcc#1 with a small amount of C and fcc#2 with a large amount of C. Then a carbide rich fcc phase should be fcc#2 and not fcc#1 even if fcc#1 is not stable.

```
 subroutine checkdefcon(lokics,lokjcs,fit,ceq)
! check if composition of lokics fits default constitution in lokjcs
! return TRUE if lokics fits default in lokjcs
! NOTE lokics and lokjcs can be the same!!
   integer lokics,lokjcs,fit
   type(gtp_equilibrium_data), pointer :: ceq
 subroutine shiftcompsets(ceq)
! check phase with several composition sets if they should be shifted
! to fit the default constitution better
! IGNORE UNSTABLE COMP.SETS
   type(gtp_equilibrium_data), pointer :: ceq
 subroutine copycompsets(iph,ics1,ics2,ceq)
```

```
! copy constitution and results from ic2 to ic1 and vice versa
   integer iph,ics1,ics2
   type(gtp_equilibrium_data), pointer :: ceq
 subroutine copycompsets2(lokph,ics1,ics2,ceq)
! copy constitution and results from ic2 to ic1 and vice versa
   integer lokph,ics1,ics2
   type(gtp_equilibrium_data), pointer :: ceq
 subroutine shiftcompsets2(lokph,ceq)
! check if the composition sets of phase lokph
! should be shifted to fit the default constitution better
   integer lokph
   type(gtp_equilibrium_data), pointer :: ceq
```

8.18.10 Select composition set for stable phase, maybe not used

```
 subroutine select_composition_set(iph,ics,yarr,ceq)
! if phase iph wants to become stable and there are several composition sets
! this subroutine selects the one with default composition set that fits best.
! For example if an FCC phase that could be an austenite (low carbon content)
! or a cubic carbo-nitride (high carbon or nitrogen content, low vacancy)
! Less easy to handle ordered phases like B2 or L1_2 as ordering can be
! in any sublatittice ... option B and F needed
   implicit none
   TYPE(gtp_equilibrium_data), pointer :: ceq
   double precision, dimension(*) :: yarr
   integer iph,ics
```

8.19 Unfinished things

There are many unfinished parts among the other sections, in particular the save/read section.

8.20 TP functions

TP functions are a sub package inside GTP. These can store simple expressions depending on T and P including unary functions LN and EXP (and maybe more). They can return a function value and the first and second derivative with respect to T and P. There are three datatypes associated with these functions described in section 4.

8.20.1 Initiate the TP fun package

```
 SUBROUTINE tpfun_init(nf,tpres)
! allocate tpfuns and create a free list inside the tpfuns
   implicit none
```

```
   integer nf
! use tpres declared externally for parallel processing
  TYPE(tpfun_parres), dimension(:), pointer :: tpres
```

8.20.2 Utilities

Because freetpfun is private.

```
 integer function notpf()
! number of tpfunctions because freetpfun is private
   implicit none
```

8.20.3 Find a TP function

```
  SUBROUTINE find_tpfun_by_name(name,lrot)
! returns the location of a TP function
! if lrot>0 then start after lrot, this is to allow finding with wildcard *
    implicit none
    integer lrot
    character name*(*)
  SUBROUTINE find_tpfun_by_name_exact(name,lrot,notent)
! returns the location of a TP function, notent TRUE if not entered
    implicit none
    integer lrot
    logical notent
    character name*(*)
```

8.20.4 Evaluate a TP function

Calculate the value and first and second derivatives.

```
 subroutine eval_tpfun(lrot,tpval,result,tpres)
!    subroutine eval_tpfun(lrot,tpval,symval,result)
! evaluate a TP function with several T ranges
!   implicit double precision (a-h,o-z)
   implicit none
   integer lrot
   double precision tpval(2),result(6)
   TYPE(tpfun_parres), dimension(:), pointer :: tpres
```

8.20.5 List the expression of a TP function

```
 subroutine list_tpfun(lrot,nosym,str)
```

```
! lists a TP symbols with several ranges into string str
! lrot is index of function, if nosym=0 the function name is copied to str
   implicit none
   character str*(*)
   integer nosym,lrot
 subroutine list_all_funs(lut)
! list all functions except those starting with _ (parameters)
   implicit none
   integer lut
```

8.20.6 List unentered TP functions

For use when reading from a TDB file as functions may call each other.

```
 subroutine list_unentered_funs(lut,nr)
! counts and list functions with TPNOTENT bit set if lut>0
   implicit none
   integer lut,nr
```

8.20.7 Compiles a text string as a TP function

Very messy

```
 SUBROUTINE ct1xfn(string,ip,nc,coeff,koder,fromtdb)
!...compiles an expression in string from position ip
!     it can refer to T and P or symbols in fnsym
!     compiled expression returned in coeff and koder
!
! >>> this is very messy
!
!...algorithm for function extraction
! 10*T**2 -5*T*LOG(T) +4*EXP(-5*T**(-1))
!
! AT LABEL 100 start of expression or after (
! sign=1
! -, sign=-1                              goto 200
! +, skip
!
! AT LABEL 200 after sign
! if A-Z                                  goto 300
! if 0-9, extract number                  goto 400
! (                                       goto 100
! ;                                       END or ERROR
! empty                                   END or ERROR
```

```
! anything else                              ERROR
!
! AT LABEL 300 symbol
! if T or P, extract power if any incl () goto 400
! unary fkn? extract (                      goto 100
! symbol                                    goto 400
!
! AT LABEL 400 after factor
! -, sign=-1                                goto 200
! sign=1
! +, skip                                   goto 200
! )                                         goto 400
! ** or ^ extract and store power incl ()   goto 400
! *                                         goto 200
! empty                                     goto 900
!
! for TDB compatibility skip #
!
! allow unary functions ABOVE(TB) and BELOW(TB) where TB= is the break temp
! check consistency
   implicit none
   integer ip,nc,koder(5,*)
   character string*(*)
   double precision coeff(*)
   logical fromtdb
 subroutine ct1getsym(string,ip,symbol)
!...extracts an symbol
!   implicit double precision (a-h,o-z)
   implicit none
   integer ip
   character string*(*),symbol*(*)
 subroutine ct1power(string,ip,ipower)
!...extracts an integer power possibly surrounded by ( )
   implicit none
   integer ip,ipower
   character string*(*)
 subroutine ct1mfn(symbol,nranges,tlimits,lokexpr,lrot)
!...creates a root record with name symbol and temperature ranges
! highest T limit is in tlimits(nranges+1)
!   implicit double precision (a-h,o-z)
   implicit none
   integer nranges,lrot
   character*(*) symbol
   TYPE(tpfun_expression), dimension(*) :: lokexpr
   real tlimits(*)
 subroutine ct2mfn(symbol,nranges,tlimits,lokexpr,lrot)
```

```
!...stores a TPfun in an existing lrot record with name symbol
! and temperature ranges, highest T limit is in tlimits(nranges+1)
   implicit none
   integer nranges,lrot
   character*(*) symbol
   TYPE(tpfun_expression), dimension(*) :: lokexpr
   real tlimits(*)
 subroutine ct1mexpr(nc,coeff,koder,lrot)
!...makes a datastructure of an expression. root is returned in lrot
!   implicit double precision (a-h,o-z)
   implicit none
   integer nc,koder(5,*)
   TYPE(tpfun_expression), pointer :: lrot
   TYPE(tpfun_expression), pointer :: noexpr
   double precision coeff(*)
 subroutine ct1efn(inrot,tpval,val,tpres)
!...evaluates a datastructure of an expression. Value returned in val
!     inrot is root expression tpfunction record
!     tpval is valuse of T and P, symval is values of symbols
! first and second derivatives of T and P also calculated and returned
! in order F, F.T, F.P, F.T.T, F.T.P, F.P.P
!
! if function already calculated one should never enter this subroutine
!
! It can call "itself" by reference to another TP function and for
! that case one must store results in levels.
   implicit none
   double precision val(6),tpval(*)
   TYPE(tpfun_expression), pointer :: inrot
   TYPE(tpfun_parres), dimension(:), pointer :: tpres
```

8.20.8 Utility to list a TP function

```
 subroutine ct1wfn(exprot,tps,string,ip)
!...writes an expression into string starting at ip
!     lrot is an index to an tpexpr record
!   implicit double precision (a-h,o-z)
   implicit none
   character tps(2)*(*)
   character string*(*)
 subroutine ct1wpow(string,ip,tps,mult,npow)
!...writes "ips" with a power if needed and a * before or after
!   implicit double precision (a-h,o-z)
   implicit none
   integer ip,mult,npow
```

```
   character string*(*),tps*(*)
```

8.20.9 Enter a TP function interactively

```
 subroutine enter_tpfun_interactivly(cline,ip,longline,jp)
! interactive input of a TP expression
!   implicit double precision (a-h,o-z)
   implicit none
   integer ip,jp
   character cline*(*),longline*(*)
```

8.20.10 Enter a dummy TP function

```
 subroutine enter_tpfun_dummy(symbol)
! creates a dummy entry for a TP function called symbol, used when entering
! TPfuns from a TDB file where they are not in order
   implicit none
   character*(*) symbol
```

8.20.11 Enter a TP function

```
 subroutine enter_tpfun(symbol,text,lrot,fromtdb)
! creates a data structure for a TP function called symbol with several ranges
! text is whole expression
! lrot is returned as index.  If fromtdb is FALSE and lrot<0 it is a new
!                             expression for an old symbol
! if fromtdb is TRUE references to unknown functions are allowed
! default low temperature limit is 298.16; high 6000
   implicit none
   integer lrot
   character*(*) text,symbol
   logical fromtdb
```

8.20.12 Some more utilities for TP function

One handles nested TP function, other abbreviated names.

```
 subroutine nested_tpfun(lrot,tpval,nyrot)
! called from ct1efn when a it calls another TP function that must be
! evaluated.  nyrot is the link to the ct1efn in the correct range
!   implicit double precision (a-h,o-z)
   implicit none
   integer lrot
```

```
   double precision tpval(2)
   TYPE(tpfun_expression), pointer :: nyrot
! use lowest range for all T values lower than first upper limit
! and highest range for all T values higher than the next highest limit
! one should signal if T is lower than lowest limit or higher than highest
! used  saved reults if same T and P
 logical function compare_abbrev(name1,name2)
! returns TRUE if name1 is an abbreviation of name2
! termintaes when a space is found in name1
! each part between _ or - can be abbreviated from the left
! case insensitive. Only 36 first characters compared
   implicit none
   character*(*) name1,name2
 subroutine below_t0_calc(t0,tpval,fun)
! calculates exp(20(1-t/t0))/(1+exp(20(1-t/t0))
! At t<<t0 K function is unity, at t>>t0 function is zero
   implicit none
   double precision t0,tpval(2),fun(6)
 subroutine above_t0_calc(t0,tpval,fun)
! calculates exp(20(1-t/t0))/(1+exp(20(1-t/t0))-1, at t>>t0 it is unity
   implicit none
   double precision t0,tpval(2),fun(6)
```

8.20.13 Routines for optimizing coefficients and some other things

They are utility routines for the assessment program to manipulate the coefficients in the models.

```
 subroutine enter_optvars(firstindex)
! enter variables for optimization A00-A99
   implicit none
   integer firstindex
 subroutine find_tpsymbol(name,type,value)
! enter variables
   implicit none
! type=0 if function, 1 if variable, 2 if optimizing variable
   integer type
   character name*(lenfnsym)
   double precision value
 subroutine enter_tpconstant(symbol,value)
! enter variables
   implicit none
   character symbol*(lenfnsym)
   double precision value
 subroutine change_optcoeff(lrot,value)
```

```
! change value of optimizing coefficient.  lrot is index
   implicit none
   integer lrot
   double precision value
 subroutine get_value_of_constant_name(symbol,lrot,value)
! get value (and index) of a TP constant.  lrot is index
   implicit none
   integer lrot
   character symbol*(*)
   double precision value
 subroutine get_value_of_constant_index(lrot,value)
! get value of a TP constant at known lrot
   implicit none
   integer lrot
   double precision value
 subroutine get_all_opt_coeff(values)
! get values of all optimizing coefficients
   implicit none
   double precision values(*)
 subroutine delete_all_tpfuns
! delete all TPFUNs.  No error if some are already deleted ...
! note: tpres is deallocated when deleting equilibrium record
```

8.20.14 Saving and reading unformatted TP funs

These are attempts to save and read TP functions from an unformatted file.

```
 subroutine tpfunsave(lut,form)
! save tpfundata on a file, unfinished
!   implicit double precision (a-h,o-z)
   implicit none
   integer lut
   logical form
 subroutine save1tpfun(lut,form,jfun)
! save one tpfun (a parameter) with index jfun on a file
!   implicit double precision (a-h,o-z)
   implicit none
   integer lut,jfun
   logical form
 subroutine read1tpfun(lut,jfun)
! read one unformatted tpfun (a parameter) with index jfun
!   implicit double precision (a-h,o-z)
   implicit none
   integer lut,jfun
 subroutine tpfunread(lin,skip)
```

```
! read tpfundata from a file, if skip TRUE ignore if function already entered
   implicit none
   integer lin
   logical skip
```

8.20.15 Create a name for optimization variables A00 to A99

Just a simple routine.

```
  subroutine makeoptvname(name,indx)
    implicit none
    character name*(*)
    integer indx
```

9 Summary

Thats all!

References

[07Luk] H L Lukas, S G Fries and B Sundman, *Computational Thermodynamics, the Calphad method*, Cambridge univ press (2007)

[15Sun1] B Sundman, U Kattner, M Palumbo and S G Fries, OpenCalphad - a free thermodynamic software, Integrating Materials and Manufacturing Innovation, **4**:1 (2015), open access