How to Write a SHARC Interface to a Quantum Chemistry Program

Hans Georg Gallmetzer, Sascha Mausenberger and Sebastian Mai

July 28, 2025

1 Introduction

SHARC interfaces are Python modules/classes that provide the SHARC dynamics driver (and also other programs within the SHARC package) with results of electronic structure calculations or potential energy surface models—i.e., properties of electronic states as a function of the molecular geometry. In particular, the largest subfamily of SHARC interfaces connects ab initio electronic structure programs (i.e., quantum chemistry, or QC programs) with SHARC. While there exists already a number of such SHARC interfaces with various QC programs, there can be multiple reasons to develop a new interface. In this document, we provide a work-through for the development of such a SHARC–QC interface. This document should also be useful for developers of fast or hybrid interfaces, although these require some additional considerations (e.g., how to avoid file I/O, efficient communication, or handling child interfaces).

The naming convention for SHARC interfaces is `SHARC_<PROGRAM_NAME>.py`. It is important to follow this convention, because SHARC tries to automatically find all available SHARC interfaces by looking for files that follow this convention. Only Python programs in `$SHARC/bin/` whose name starts with "`SHARC_`" will be found correctly. The string `<PROGRAM_NAME>` must be all uppercase, but can contain underscores.

This document is organized as follows:

1. In this introduction, we introduce briefly the template code for SHARC–QC interfaces, listing the different functionalities that need to be implemented in general.
2. In the next section, we lead the developer through the design process, making them aware of several important aspects of QC software. The section also discusses what kind of QC input the interface will eventually need to generate and what has to be read from the output files.
3. In the next section, we briefly summarize how the implementation can be tackled.
4. The subsequent section discusses the implementation in greater detail.
5. Finally, in the last section, we provide a glossary of important aspects of SHARC interfaces.

Writing an interface is no easy task. We provide a simple example interface code in `$SHARC/SHARC_BASICORCA.py` to give an overview how interfaces are build. Additionally, we provide in `$SHARC/../examples/SHARC_ABINITIOTEMPLATE.py` the empty skeleton code as a starting point in developing an interface. The attributes and methods in these two files are organized in the following way:

- **Infos section**: General information about the interface.
- **Template/Resource section**: Two input files needed to run a SHARC-calculation
- **Standard methods section**: Incorportation of the Infos section into the class definition.
- **Initialization section**: Methods that prepare the interface for it's use right after it is started.
- **Run section**: Manages preparation and execution of QC program.
- **Scheduling**: Manages job scheduling if QC program needs more than one calculation per timestep.
- **Data collection section**: Outputparsing of QC program.
- **Setup section**: SHARC provides additional tools for setting up different types of calculations (single points, trajectories, optimisations). These methods are necessary for the communication with these tools.
- **Additional methods section**: Place to put all of the additional methods that you need to implement in the interface.
- **Main function section**: No need to touch this.

2 Execution of an interface

Currently, there are two ways a SHARC interface is used for dynamics. Traditionally, SHARC dynamics is handled by calling `sharc.x`. This approach uses file-based communication via the `QM.in` and `QM.out` files. In newer SHARC versions (from 2.1 onwards) SHARC dynamics can also be run via PySHARC (in SHARC 4, this is done using `driver.py`) which doesn't need file-based communications anymore. For fast methods, e.g., semiempirical methods or model potentials, this yields a substantial speedup over a simulation trajectory. For more information, please look up Chapter 6 of the SHARC manual. Both communication schemes are depicted in Figure 1.

There are two additional ways to use the interface. First, you can call the interface like any other Python script by executing this command:

```
$SHARC/SHARC_INTERFACENAME.py QM.in
```

This is useful for single point calculations, initial vertical excitation calculations, optimizations, scans, etc. This requires that the QM.in file complies with the specifications (Chapter 6 of the manual). Also, `INTERFACENAME.resources` and `INTERFACENAME.template` need to be provided in the same folder as QM.in, with all the necessary keywords. If your interface works correctly you will get a QM.out file as an output.

The last way your interface can be used is through the setup scripts in `bin/`. Here, the interface object participates in the set up of the simulations.

3 The planning stage

Before starting to code up your interface, typically there are several aspects of the interface that should be planned properly. This avoids unnecessary changes later in the development. This section lists several important aspects of QC programs and how they affect the SHARC interface. We will use the `SHARC_BASICORCA.py` interface as an example.

3.1 General aspects

Quantum chemistry program Obviously, the first design decision is about the QC program, which will affect all the points—available methods, quantitites, etc—discussed below. Here, we want to emphasize that also the usage of the

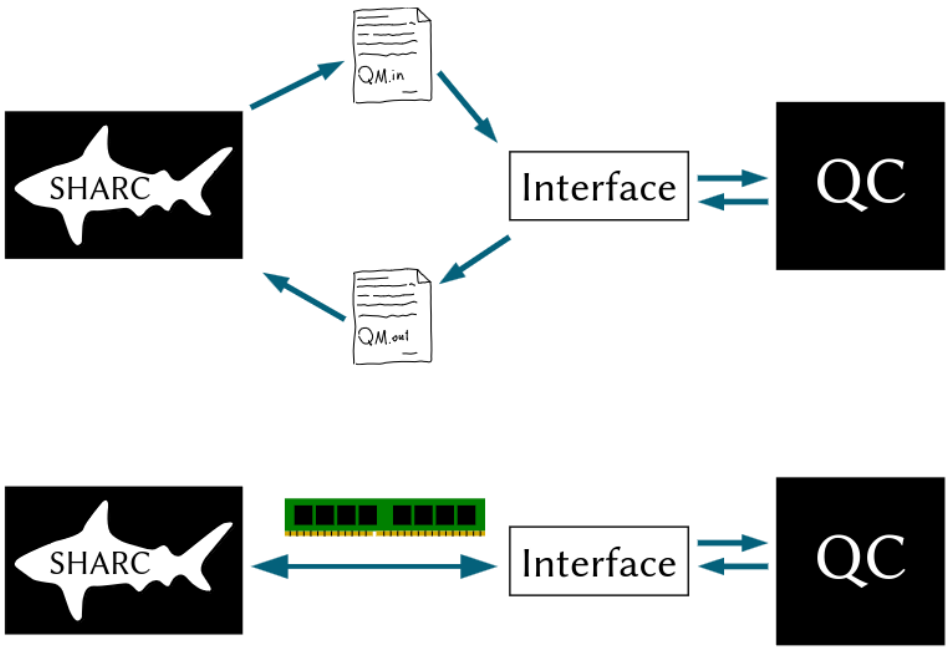

Figure 1: Upper portion: file-based communication. Lower portion: memory-based communication.

chosen program is important:

- **How is the program executed?** Most QC programs are simply started with a system call and an input file. There are a few QC programs with server–client models, where the QC program runs continuously as a server and accepts input via message passing. Furthermore, some QC programs have Python (or other) APIs, from where they can be called directly. Make sure that the chosen execution method is compatible with everything you want to do.
- **How is the input syntax of the quantum chemistry program?** Many QC programs use a declarative, job-based input format (e.g., Gaussian, ORCA, ADF), which simply specifies the system and the tasks to do, and leave the execution order and program logic to the QC program. Other QC programs use folders with several input files and/or come with input generators (e.g., Turbomole, Columbus) and might require calling different modules in a particular order. In yet other QC programs, the input files are akin to imperative computer code, where different steps are executed in order, possibly with loops and conditionals (e.g., Molpro, Molcas).
- **How are the output results obtained?** Many QC programs simply provide a log file with all results, which will need to be parsed from ASCII and which have limited accuracy. Some QC programs write results to several separate files in ASCII or binary formats. Yet some other QC programs can output structured or binary files (e.g., JSON, XML, HDF5), which are easier to parse. The easiest way of obtaining the outputs is through dedicated APIs, although they must be compatible with everything you want to do.
- **Basis functions and wave function/density matrix representation?** Some advanced features in SHARC (wave function overlaps, AO overlap integrals, RESP fitting, ECI) use PySCF to compute various integrals. This requires that the results from the QC program can be converted into a format that is compatible with PySCF. This involves, e.g., basis function definitions (contraction coefficients, normalization, AO order, spherical/Cartesian, order of angular functions, ...), whether the wave function can be represented as a configuration interaction function based on Slater determinants, and/or whether density matrices are available.

Electronic structure methods Depending on what the QC program is used for you need to take care of different things.

- **For which kind of applications is the interface needed?** Depending on whether you want to only perform single-points (e.g., for spectra simulation, LVC parametrization) or dynamics, different quantities are needed.
- **Do you only want to use one specific method of the QC program or should the user be able to choose between several different methods?** In the second case the interface needs to be general enough to take care of all the different methods.
- **Is the method used a single- or multi-reference method?** Single-reference methods make a distinction between reference state(s) and excited states, which are not treated on the same footing. Multi-reference methods typically treat all states at the same footing. This distinction governs, e.g., whether ground state–excited state conical intersections are correctly described. It also affects the interface code, e.g., how energies are computed (total energy vs. ground state energy plus excitation energy). Furthermore, multi-reference methods often require more effort to obtain smooth PESs (e.g., consistent orbital spaces, reference states, state-averaging).
- **What kind of multiplicities can the QC program deal with?** Single-reference methods often rely on restricted/unrestricted reference determinants. They require different "ground states" for different multiplicities (and thus separate calculations) and unrestricted calculations can suffer from spin contamination. For many single-reference methods/programs, treating only singlets or singlets+triplets (from a restricted singlet reference) is easiest. Multi-reference methods typically can treat any multiplicity and do not suffer from spin contamination.

Electronic structure quantities SHARC can deal with different types of properties and depending on the calculations you want to run it needs certain information. So it is important to know what data your QC program can give to SHARC:

- Which electronic structure quantities can be provided (i.e., which **requests** can be served)? Some of the most important requests in SHARC are: energies for each state, dipole moments for each state, transition-dipole moments for some/all pairs of states, gradients for each state, spin-orbit couplings (SOCs) between the states. See the SHARC manual in Section 6.0.1 for a complete list.
- Do you have a way to compute electronic couplings? The traditional quantity here is the non-adiabatic coupling (NACs) vector. However, not all QC programs/methods can provide NAC vectors. If they are available, they are typically expensive and using them in dynamics might require short time steps. Alternatively, one can use wave function overlaps, which are explained in Section 5.3.

- How can all the required quantities be produced (i.e., what input needs to be written, what needs to be read out, is it necessary to run the QC program more than once)?

3.2 Detailed planning

Once the general design decisions have been made, it can be useful to also plan out the precise capabilities of the interface.

Electronic structure settings Every SHARC QC interface uses a so-called **template** file that defines all the electronic structure settings of the calculation. It is useful to make a draft of such a template file to know all the possible settings that need to be considered later. These might include:

- The different possible electronic structure methods. If your interface can only do one method, this point is not needed.
- Basis sets. In the simplest case, this is one keyword for the atomic basis set of all atoms. However, you might also need various auxiliary basis sets or want to control the basis sets for each element or atom separately. Finally, sometimes it is important to know whether Cartesian and/or spherical basis functions can be used.
- Settings for DFT calculations. These includes exchange-correlation functionals, settings for the integration grids, usage of the Tamm–Damcoff approximation, dispersion correction, etc. You might want to check whether all your desired quantities are available with all of these possible settings.
- Settings for multi-reference methods. These include active space specifications (number of active electrons, inactive and active orbitals, RASSCF settings), state-averaging settings (number of states, possibly per multiplicity), or settings for post-CASSCF methods (e.g., various shifts for CASPT2), etc.
- General settings. These include implicit solvent treatment, relativistic effects, and acceleration methods like resolution of identity/Cholesky decomposition/density fitting, settings for Davidson diagonalization, etc.
- Verbatim input. In some cases, it is useful to allow the user to specify arbitrary input that will be pasted verbatim into the generated QC input files. This works easiest with simple declarative input formats where there is no ambiguity where the verbatim block should be placed. This is an expert option.

Note that not all of these options are required. You make the decision which options you want to incorporate into the interface. Most options can also easily be added later. If some of these options affect how the output is formatted, then you should include these options right from the start.

We provide an exemplary template file in `$SHARC/../examples/SHARC_BASICORCA/BASICORCA.template`:

BASICORCA.template

```
##Basis set for the ORCA calculation
basis 6-31G

##Functional for the ORCA calculation
functional PBE
```

This template file can be directly parsed with the template parsing code of the base interface class `SHARC_INTERFACE.py`. Hence, if you use this format, you do not need to write any code to parse your settings. The template file format is based on the key-value principle. First, a key is provided to tell the interface which template variable is being addressed, followed by the value of this template variable. In the aforementioned file, all two template keys are addressed. First, the basis key is set to the 6-31G basis set. Lastly, the functional key is set to PBE. If a template key is not defined in the template file, then the standard value, defined in the interface script, will be used. It is important to ensure that the type of value corresponds to the types defined in the interface file. For example, you cannot enter a boolean as the basis set for the calculation; it must be a string.

Computational resources (Nearly) every SHARC QC interface also uses a **resource** file that defines all settings that do not affect the electronic structure results, but is nonetheless important for the calculation. It makes sense to also plan the content of this resource file before coding up the interface:

- Path to the QC installation. If the interface should call the QC program via a system call, the interface needs to know where the QC program is installed. Providing the path explicitly in the resource file is generally better than relying on the system search path. If the interface knows the installation path, it might even automatically set up all necessary environment variables for the user. If the QC program is called via server-client models or APIs, this might not be needed.

- Path to a scratch directory. Most QC programs rely on file I/O to run. It is not a good idea to let this happen in the folder where the interface is executed (a subdirectory of a SHARC trajectory). Instead, a scratch directory can be used to run the QC program. This is useful because in this way the temporary QC files are more easily be deleted later, and a fast file system can be used if the QC program does heavy disk I/O. A scratch directory is used by all current SHARC QC interfaces.
- Number of CPU cores and main memory. Many QC programs can run in parallel. Putting these options in the resource file makes it possible for the user to control how much resources to use in a calculation.
- Settings for wave function overlap calculations. If your interface should be able to calculate wave function overlaps, then several settings need to be provided to the interface (path to the overlap code, truncation threshold for the wave functions, frozen core settings, etc).
- Settings for wave function analysis in TheoDORE. If your interface should use TheoDORE for wave function analysis, then several settings need to be provided to the interface (path to TheoDORE, fragmentation scheme, requested descriptors, etc).
- Orbital guess settings. Most SHARC QC interfaces use user-provided orbitals or orbitals from a previous time step as input guess for SCF/MCSCF calculations. Some of these interfaces allow users to change this default behaviour.
- Debug-related settings. Some SHARC QC interfaces have keywords to provide extra output, to skip the clean-up of the scratch directory, or to skip the QC program execution (and only read out results from an existing scratch directory).

We provide an exemplary template file in `$SHARC/../examples/SHARC_BASICORCA/BASICORCA.resources`:

BASICORCA.resources

```
## Directory to the ORCA program
orcadir /usr/license/orca5/orca_5_0_4_linux_x86-64_shared_openmpi411

## Directory to the scratch directory
scratchdir $TMPDIR
```

As for template files, this file can be parsed directly by already-provided interface base class code. The resources file format is based on the key-value principle. First, a key is provided to tell the interface which template variable is being addressed, followed by the value of this resources variable. In the aforementioned file, all two resources keys are addressed. First, the orcadir key is set to the path that leads to the orca executable. Second, the scratchdir key is set to the scratch directory. If a resources key is not defined in the resources file, then the standard value, defined in the interface script, will be used. It is important to ensure that the type of value corresponds to the types defined in the interface file. For example, you cannot enter a boolean as the scratchdir of the system; it must be a string.

Job scheduling Scheduling is important for programs that need several different runs in order to yield all the needed data.

- Are gradients needed for more than one state? For instance some programs can only calculate the gradients for one state per run. So if gradients for several states are needed more calculation runs are needed.
- Can your program calculate gradients and NACs separately? If yes, then you can parallelize these calculations and be more efficient.
- Can more than one multiplicity be computed at the same time? Some programs can calculate different multiplicities but only in separate runs.

The example interface `SHARC_BASICORCA.py` does not use the job scheduler that is implemented in the ab initio base class. However, the main idea when using the scheduler is to implement a routine `execute_from_qmin()` that does exactly one QC call based on a `QMin` object. One then generates in a preceeding step a list of lists of `QMin` objects for the different individual jobs, and then lets the scheduler execute this list.

Input writing At this point, you should have a good idea of all the possible/desired requests, options, and settings that can affect your QC calculation. These various pieces of input will be collected in the `QMin` object of your interface class (filled from reading a `QM.in` file or getting called by a caller code, reading the template file, and reading the resource file). Based on this information, you should be able to manually write input files for your QC program that can generate all the desired results.

For example, the `SHARC_BASICORCA.py` interface produces this input file with the PBE/6-31G method and requests to compute S_0 plus two excited singlets and the gradient of the S_0:

BASICORCA.inp

```
! 6-31G PBE engrad
%tddft
        tda false
        nroots 2
        sgradlist 0
end

%output
        Print[ P_MOs ] 1
end

%coords
        Ctyp xyz
        units bohrs
        charge 1
        mult 1
        coords
        C          0.000000000      0.000000000      0.000000000
        N          0.000000000      0.000000000      2.551130270
        H          1.782204489      0.000000000     -1.028955876
        H          1.782204489      0.000000000      3.551740254
        H         -1.782204489      0.000000000      3.551740254
        H         -1.782204489      0.000000000     -1.028955876
        end
end
```

Output parsing Once you are able to write input files for all relevant cases, you should run them in your QC program and figure out how the output is formatted and where all desired quantities are found. Typically, this includes the standard out/log file of the QC program, but sometimes also specialized files in various formats are available.

The parsing functions for all of the needed methods to use wave function overlaps can be found in the **Additional Methods** section of the `$SHARC/bin/SHARC_BASICORCA.py` file.

For example in the ORCA output file, `BASICORCA.log`, the energies of each state can be found:

BASICORCA.log

```
----------------
TOTAL SCF ENERGY
----------------

Total Energy       :          -94.82055220 Eh            -2580.19840 eV
...
---------------------------------
TD-DFT EXCITED STATES (SINGLETS)
---------------------------------

the weight of the individual excitations are printed if larger than 0.01

STATE  1:  E=   0.275825 au      7.506 eV    60536.6 cm**-1 <S**2> =   0.000000
     6a ->   8a  :     0.999051

STATE  2:  E=   0.366944 au      9.985 eV    80534.8 cm**-1 <S**2> =   0.000000
     5a ->   8a  :     0.970982
     7a ->   9a  :     0.016606
```

A method has to be implemented that ready exactly these values and compute the absolute energies of each state by adding the ground state energy and the excitation energies (the interface must report all quantities in atomic units). The output parsing must be robust enough to work even if the output changes due to different keywords. You also need to make sure that output file doesn't change if different types of calculations are made.

If you plan to implement wave function overlaps, then note that you need to be able to extract the following quantities:

- CI coefficients and orbital occupation strings,
- MO coefficients,
- AO overlap matrices (or basis set information consistent with PySCF conventions).

Running excited state dynamics with overlaps instead of NACs allows for bigger time-steps and is not as prone to miss conical intersections. Furthermore, NAC calculations are also very expensive. For more information see Section 5.3.

4 The implementation stage

If you first want to know more about the structure of the relevant objects, read further in Section 5. If you want to start coding right away, read further here.

Start by copying the interface template from `$SHARC/../examples/SHARC_ABINITIOTEMPLATE.py` to a file corresponding to your QC program and call it `$SHARC/SHARC_<INTERFACENAME>.py`. Implement the interface in this order. Besides the required methods, you can make as many "private" methods as you want.

1. **Infos section**: Here you have to give some general information. Write the name of your interface, fill out the attributes below, and add all the supported features/requests (Section 5.1.6) of your interface to the `all_features` set. The all_features set defines which requests the interface can fulfill (i.e., which electronic properties the interface can produce) and how it behaves (e.g., when passed point charges for electrostatic embedding).
2. **Template/Resource section**: Most interfaces use a set of two input files to specify how to perform the calculation. The template file contains all QC-related information (e.g., functional, basis set, active space, accuracy-related settings) and is generally identical for all trajectories in the same project. The resources file contains all computing-related information (i.e., paths, parallelization, memory), which might differ between trajectories of the same project. Define class variables specific to your interface and add all of the necessary resources and template key with the standard values and correct type.
3. **Standard methods section**: These methods incorporate the Infos section into the class definition. Change `SHARC_<INTERFACENAME>` and make sure to be consistent over the whole interface file.
4. **Initialization section**: The initialization stage refers to all preparation work that the interface needs to do once, after the interface is started. This includes reading the system specifications (number of atoms/point charges, elements, number of states), template, and resources. It also refers to setting up a number of arrays and dictionaries that describe these data (e.g., a list of the state's quantum numbers). All the information is collected into a QMin object.
    - `read_template()` is usually trivial, but you need to care for inputs that can cause conflicts, think of what inputs should be allowed. Besides code to parse special keywords, the base class code usually provides everything needed.
    - `read_resources()` also is usually trivial. Put in the paths to the needed programs, number of cores, memory etc. Besides code to parse special keywords, the base class code usually provides everything needed.
    - `setup_interface()` cares for all the checks that are "static", i.e., do not depend on the current requests. This function can be used to set up maps (i.e., lists or dictionaries that provide information about the electronic states, e.g., ionmap and gsmap) but also to build the jobs dictionary, when more than one calculation run per step is needed. This method takes care of the things that need to be done after you get the additional information from the template and resources file.

    For more information see Section 5.1.1.
5. **Run section**: Each calculation step consists of (i) receiving the current coordinates, step information, and property requests, and (ii) executing the QC program. Depending on how your program works you have to give it all the information needed (write the input file and execution command). This section might also contain error handling if jobs fail, and code to handle files that need to be saved or deleted. Some data needs to be updated for each step of a simulation of a trajectory.
    - `read_requests()` receives the requests. Normally you don't have to change anything here.
    - `set_coords()` receives the new coordinates. Normally you don't have to change anything here.
    - Implement the `run()` method, it runs the QC program and you can call auxiliary programs like WFoverlap (Section 5.3) or TheoDORE. You need to take care of the files that need to be saved in savedir and restart files (initial guess, molden, orbital mapping etc.). If several calculations are needed for one timestep you can optionally do this via scheduling (by implementing `execute_from_qmin`).
6. **Scheduling section**: Making a schedule is convenient if you need to make several calculations for one time step. Run the job schedule and care for the files that need to be saved for later and for restarting. We have already written an extra method called `_gen_schedule()` which you can use if needed. Scheduling is useful if your program needs several runs to get all of the needed information, e.g.:
    - If your program can only calculate the gradient for one state in a single calculation run you need to schedule separate runs for each gradient.
    - Different multiplicities can sometimes also only be calculated in separate runs.

Basically, you write a list of dictionaries, called schedule, with a `QMin` object as value for each sub-job. The schedule is executed via `runjobs(schedule)` which in turn executes every sub-job via `execute_from_qmin()`. `execute_from_qmin()` calls the QC program, so you have to give it all the information needed (write the input file and execution command). Get saved information from previous time steps or schedule jobs of the same time step. If you need to generate restart files by hand this can be done with the `create_restart_files()` method.

7. **Data collection section**: Here, the results of all the QC calculations is collected via parsing functions. The results are stored into a QMout object, which is eventually returned to the caller of the interface. Implement the `getQMout()` method in order to fill the `QMout` object with the necessary data like the energies of the calculated state, transition dipole moments etc. For more information see Section 5.2.
8. **Setup section**: The SHARC interfaces are also responsible to cooperate with the setup scripts of the SHARC package. These routines implement communicating the available features, doing an input dialogue with the user, and setting up a directory with all interface-related files.
    - `get_features()`: does not need to be changed normally (for ab initio interfaces that have static feature lists). It is more involved for hybrid interfaces that combine several interfaces. It handles conflicts between requests for different sub-interfaces that are managed by the hybrid interface. You can exclude features that are not supported by the setup scripts.
    - `get_infos()`: Check if the user already has a template and resources file. If not then fill the `INFOS` dictionary with all the information that is needed for the template and resources files through command line inputs.
    - `prepare()`: Prepares the work directory according to information of the `INFOS` dictionary and local information (`self.setupINFOS`).
9. **Additional methods section**: Here you can put all of the additional methods that you need like parsing methods for the output files. Many QC interfaces can make use of SHARC's wfoverlap code or the TheoDORE wave function analysis tool set and call these programs during the run section.
10. **Main function section**: Here lies the call to the main function. No need for changes.

4.1 Template and Resources Definition/Methods:

We are using the `SHARC_BASICORCA.py` interface as an example. The template and resources files allow the user to control the QC program. The possible template and resource variables can be defined in the `Template/Resources Definition` section. As these files are read as key-value pairs, you must define the key (the variable name as a string) together with its standard value in the `self.QMin.template/resources.update()` dictionary and the type of value that can be entered in the `self.QMin.template/resources.types.update()`.

```
# ------------------------------| Template/Resources Definition |----------------------------------

self._need_this_later = None

self.QMin.resources.update(
    {
        "orcadir": None, # Path to the executable of the QC-program
    }
)
self.QMin.resources.types.update(
    {
        "orcadir": str,
    }
)

self.QMin.template.update(
    {
        "basis": "6-31G",
        "functional": "PBE",
    }
)
self.QMin.template.types.update(
    {
        "basis": str,
        "functional": str,
    }
)
```

In the `Initialization` section, the `read_template()` and `read_resources()` methods are called to read the respective template and resources files and import their values. You must ensure that certain input combinations do not cause conflicts, and that an error is thrown to make the user aware of these problems if they do. The `setup_interface()` method, on the other hand, can be used to make adjustments to the interface that are not template- or resource-specific.

```
# ------------------------------| Initialization|-------------------------------------------------

def read_template(self, template_file: str = "BASICORCA.template", kw_whitelist: list[str] | None = None
    ) -> None:
    super().read_template(template_file, kw_whitelist)

def read_resources(self, resources_file: str = "BASICORCA.resources", kw_whitelist: list[str] | None =
    None) -> None:
    super().read_resources(resources_file, kw_whitelist)

def setup_interface(self) -> None:
    super().setup_interface()
    self.QMin.control["states_to_do"] = deepcopy(self.QMin.molecule["states"])
```

We suggest that the an interface developer starts with the Infos and Standard methods section, and then implements the code for reading the template and resource file, as seen in these examples. Once this is done, the developer can add a `print(self.QMin)` statement in the `run()` method, so that the interface is already runnable.

4.2 Writing of the Input File:

An example input generation routine is given here:

```
@staticmethod
def generate_inputstr(qmin: QMin) -> str:
    """
    Generate ORCA input file string from QMin object
    """
    job = qmin.control["jobid"]
    qmin["molecule"]["charge"][0]  #Charge is provided by SHARC, no need to define it in the template
    file.

    # excited states to calculate
    states_to_do = deepcopy(qmin.control["states_to_do"])

    # gradients
    do_grad = False
    if qmin.requests["grad"] and qmin.maps["gradmap"]:
        do_grad = True

    string = "! "
    keys = ["basis", "functional"]
    string += " ".join(qmin.template[x] for x in keys if qmin.template[x] is not None)

    string += " engrad\n" if do_grad else "\n"
    #Excited states
    if max(states_to_do) > 0:
        string += f"%tddft\n\ttda false\n"
        string += f"\tnroots {max(states_to_do)-1}\n"
        if do_grad:
            string += "\tsgradlist " +  ",".join([str(i[1]-1) for i in qmin.maps["gradmap"]]) + "\n"

        string += "end\n\n"

    string += "%output\n"
    string += "\tPrint[ P_MOs ] 1\n"
    string += "end\n\n"

    string += "%coords\n\tCtyp xyz\n\tunits bohrs\n"
    string += f"\tcharge {charge}\n"
    string += f"\tmult 1\n"
    string += "\tcoords\n"
    for iatom, (label, coords) in enumerate(zip(qmin.molecule["elements"], qmin.coords["coords"])):
        string += f"\t{label:4s} {coords[0]:16.9f} {coords[1]:16.9f} {coords[2]:16.9f}\n"
    string += "\tend\nend\n\n"
```

```
42
43      return string
```

This `generate_inputstr()` function is defined in the `SHARC_BASICORCA.py` interface. It produces a valid ORCA input file from the data it is given by SHARC through the template, resources, and `QM.in` file. For an example, see above.

4.3 Implementation Checklists

This subsection provides a checklist in which order one could write the interface. This subsection requires that one has planned the interface (section 3).

1. Basics (attributes, features)
2. Create example templates and resources, plus an example QM.in file for testing while coding, update keywords, then run and see how data is represented in QMin object
3. finalize the set of initialization routines (especially setup_interface, add checks/warnings, etc)
4. build the run-related routines: (optionally) make a schedule, make folders, write inputs, copy files, call program
5. take care of restarting functionalities
6. build the output parsing routines
7. take care of the setup-related routines

4.3.1 Checklist for methods that need to be implemented

Some methods do not need to be changed and are the same for every interface:

- version()
- versiondate()
- changelogstring()
- read_requests()
- authors()
- name()
- description()
- about()

These are the methods that you need to care for:

- execute_from_qmin()
- read_template()
- read_resources()
- setup_interface()
- getQMout()
- run()

These are relevant for setup scripts:

- get_features()
- get_infos()
- prepare()

Since object-oriented programming is used, you can also use all the methods that are implemented in the parent classes of your interface (`SHARC_ABINITIO`, `SHARC_INTERFACE`). Before implementing a method yourself, look if it already has been written in one of the parent classes.

5 Glossary

5.1 QMin Object

All the necessary information that an interface needs can be found in three files/objects:

- `QM.in` file (or via caller)
- `INTERFACENAME.template` file
- `INTERFACENAME.resources` file

5.1.1 Template and Resources Files

Think of what your program needs to run and what additional information, that you cannot get from QMin (look up in section 5.1), is necessary. This additional information needs to be provided through an external file. The template file is used for simple additional inputs like flags, basis sets, functionals, active space etc. and must be called `INTERFACENAME.template` Some examples for the template:

```
basis 6-31G
functional PBE
act_orbs 45 46 48
```

If you need additional information that has more data like initial guesses, orbital mapping etc. you need to take care of these files through the restart logic and file saving in the run section.

In the `resource` file you can put paths, memory, number of CPU cores, initial orbital source for your QC program and also `WFoverlap`. It must be called `INTERFACENAME.resources`. Some examples for the resources:

```
programdir /home/user/programname/bin
programversion 1.0
ncpu 3
scratchdir $TMPDIR
savedir ./SAVEDIR
wfoverlap $SHARC/../wfoverlap/source/wfoverlap_ascii.x
```

5.1.2 QMin object content

The heart of your interface is the QMin object. It contains the following attributes:

- **molecule**
- **coords**
- **save**
- **requests**
- **maps**
- **resources**
- **template**
- **scheduling**
- **control**

5.1.3 molecule

- **natom**: number of atoms
- **elements**: elements names in input order
- **unit**: "bohr" or "angstrom"
- **factor**: to convert units
- **states**: list of number of requested states per multiplicity (S, D, T, Q, 5, 6, ...)
- **nstates**: sum of states over all multiplicities
- **nmstates**: total number of states, including the multiplet sublevels
- **point_charges**: Boolean, whether point charges are present
- **npc**: number of point charges

5.1.4 coords

- **coords**: Coordinates of your system in bohr.
- **pccoords**: Coordinates of the point charges in bohr.
- **pccharges**: Charges of the point charges in elementary charges.

5.1.5 save

- **savedir**: Path were interface-data is stored for later use. E.g. for the calculation of wavefunction overlaps the AO overlaps, MO coefficients, determinants and CI coefficients should be stored here.
- **step**: current step in a dynamics.
- **previous_step**: previos step index
- **init**: Boolean whether this is the first calculation
- **newstep**: Boolean whether this is a calculation at a new time step
- **samestep**: Boolean whether this another calculation at an already computed time step
- **always_guess**: Boolean whether orbitals from previous time step should be used or guessed
- **always_orb_init**: Boolean whether orbitals from previous time step should be used or provided orbitals used

5.1.6 requests

These are standard requests:

- **h**: Energies
- **soc**: Spin-orbit couplings
- **dm**: Dipole moments
- **grad**: Gradients
- **nacdr**: Non-adiabtic couplings
- **overlap**: Overlaps, can be calculated with the WFoverlap program
- **phases**: Phases, can be calculated from the overlaps
- **ion**: Dyson norms with ionic states
- **theodore**: Run TheoDORE for wave function analysis
- **socdr**: Derivatives of spin-orbit couplings
- **dmdr**: Derivatives of dipole moment matrices

These are more specialized requests that are mostly restricted to ab initio interfaces:

- **multipolar_fit**: Do a RESP fit to return partial/transition charges and possibly higher order terms
- **mol**: Basis set information for PySCF
- **density_matrices**: Sets of state/transition density matrices in the AO basis, consistent with the mol request
- **dyson_orbitals**: Dyson orbitals with ionic states
- **molden**: Molden file
- **cleanup**: remove save directory at the end
- **retain**: garbage-collect files in the save directory that are from old time steps
- **savestuff**: copy raw quantum chemistry output to save directory
- **nooverlap**: Do not save data needed for overlaps

Note that these requests have some similarity, but are not the same thing as features as they are provided by `get_features()`. These are the features in use:

- **h**
- **soc**
- **dm**
- **grad**
- **nacdr**
- **overlap**
- **phases**
- **ion**
- **theodore**
- **socdr**
- **dmdr**

- **multipolar_fit**
- **mol**
- **density_matrices**
- **point_charges**: Currently the only feature that does not directly correspond to a request. It specifies whether an interface can work with external point charges (for electrostatic embedding QM/MM).

5.1.7 maps

- **statemap**: dictionary mapping state indices to multiplicity, M_S value, and cardinal state index
- **mults**: set of requested multiplicities
- **gsmap**: dictionary assigning a ground state for each state (needed for single-reference methods like TDDFT)
- **gradmap**: set of multiplicity/state pairs to compute gradients (used to filter out multiple requests for the same multiplet gradient)
- **nacmap**: set of multiplicity/state/state tupels to compute NACs
- **densmap**: set of density matrices to be extracted
- **chargemap**: dictionary mapping state indices to charge
- **multmap**: dictionary mapping multiplicities and "jobs" in an interface that does multiple QC jobs for one time step

5.1.8 resources

Here you can access all of your resource variables.

5.1.9 template

Here you can access all of your template variables.

5.1.10 scheduling

- **schedule**: list of sets/lists of QMin objects, this object is the principal tool to drive the calling/parallelization of external ab initio code. Most programs require/favor multiple runs until all requested data is computed (e.g., multiple independent wave function computations, multiple gradients/NACs, etc).

5.2 QMout Object

You should write getter functions that parse the output files of your program. This should be fairly easy but there are some conventions that you need to follow. For instance the dimensions of the matrices of the QMout object for the energy (h), the dipole moments (dm), gradients (grad), NACs (nacdr) and the overlaps (overlap).

```
states = [S, D, T, ...]
nmstates = sum([(m+1)*i for m,i in enumerate(states)])
```

- **h** = (nmstates×nmstates) complex and Hermitian, off-diagonal elements are the SOCs
- **dm** = (3×nmstates×nmstates) real and Hermitian, off-diagonal elements are the TDMs
- **grad** = (nmstates×natom×3) real
- **grad_pc** = (nmstates×npc×3) real
- **nacdr** = (nmstates×nmstates×natom×3) real and anti-Hermitian
- **nacdr_pc** = (nmstates×nmstates×npc×3) real and anti-Hermitian
- **overlap** = (nmstates×nmstates) complex
- **phases** = (nmstates) complex, each element has norm 1

It is required to use numpy arrays because the Python–C interface assumes that.

5.3 WFoverlap

If your program can provide AO-overlaps, Slater-determinants and MO-coefficients of two consecutive time steps, you can use the WFoverlap program to run dynamics via overlaps instead of NACs, which are more expensive. We are following the convention used by PySCF in SHARC.

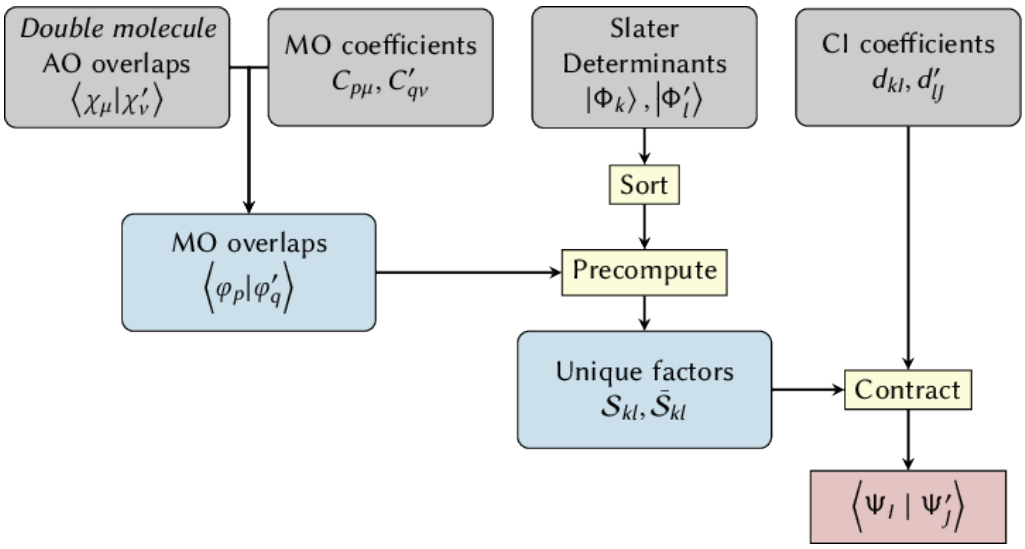

Figure 2: Workflow of the wavefunction overlap program.

AO-Coefficient File The AO-coefficient matrix between the current and the previous step needs to be written like this:

```
12 12
 1.000000000000000e+00  0.000000000000000e+00  0.000000000000000e+00  ...
 0.000000000000000e+00  1.000000000000000e+00  0.000000000000000e+00  ...
 0.000000000000000e+00  0.000000000000000e+00  1.000000000000000e+00  ...
...
```

The first line consists of the dimensions of the AO-overlap matrix which is the number of AOs times the number of AOs. Then the matrix is written into the file.

Determinant-File The determinants file has to look like this:

```
4 12 16
dddddeeeeee   0.998887000000    0.024878000000   -0.011107000000   -0.014844000000
dddddbaeeeee   0.016260627540   -0.699431844184   -0.007714534983    0.008956214491
dddddabeeeee  -0.016260627540    0.699431844184    0.007714534983   -0.008956214491
dddddedeeeee  -0.007955000000    0.074317000000    0.003208000000    0.004618000000
...
```

The first line consists of three numbers:

- the number of states (columns in the file)
- the number of MOs (length of the determinant strings)
- the number of determinants in the file (length of the file)

In the following lines the determinant string of each Slater determinant has to be written together with the CI coefficients of the determinant for each state. The string is made out of these four symbols:

- d - doubly occupied
- a - singly occupied (α)
- b - singly occupied (β)
- e - empty

MO-Coefficients File WFoverlap supports three different file formats for the MO-coefficients file: Columbus, Molcas and Turbomole. Here is an example of the Columbus format:

```
2mocoef
```

```
header
1
MO-coefficients from ABCDEFG
1
12   12
a
mocoef
(*)
-4.013668212000e-01  8.261835700000e-03 -4.350815100000e-03
-1.606350215000e-01 -8.418507263000e-01  4.908336150000e-02
-6.586301360000e-02  5.546625880000e-02 -7.046358850000e-02
...
 1.153971350000e-02 -7.408251650000e-01  6.233744710000e-02
orbocc
(*)
 0.000000000000e+00  0.000000000000e+00  0.000000000000e+00
 0.000000000000e+00  0.000000000000e+00  0.000000000000e+00
 0.000000000000e+00  0.000000000000e+00  0.000000000000e+00
 0.000000000000e+00  0.000000000000e+00  0.000000000000e+00
```

In the 6th line the number of MOs has to be written followed by the number of AOs. In the mocoef-section you have to write the MO-coefficients, first all the coefficients for the first MO are written, followed by the coefficients of the second MO. The orbocc-section can be filled with zeros, is is a vector of the length of the number of MOs written in a a 3 by N fashion.

5.3.1 Double-Molecule Trick

Sometimes it is hard to get the AO overlaps from a QC program but in some programs you can simply write the structure of your molecule from the current time step followed by the structure of the previous time step and this will give you the AO-overlap between the two molecules.