PALM - Debugging

PALM group

Institute of Meteorology and Climatology, Leibniz Universität Hannover

last update: 21st September 2015

Principal Sources of Errors

PALM runs can give rise to a large variety of errors ...

Some of the main possible reasons for errors are:

Principal Sources of Errors

PALM runs can give rise to a large variety of errors ...

Some of the main possible reasons for errors are:

- Missing or wrong options in the `mrun` call

Principal Sources of Errors

PALM runs can give rise to a large variety of errors ...

Some of the main possible reasons for errors are:

- Missing or wrong options in the `mrun` call
- Errors in the configuration file

Principal Sources of Errors

PALM runs can give rise to a large variety of errors ...

Some of the main possible reasons for errors are:

- Missing or wrong options in the `mrun` call
- Errors in the configuration file
- Errors in the NAMELIST parameter file

Principal Sources of Errors

PALM runs can give rise to a large variety of errors ...

Some of the main possible reasons for errors are:

- Missing or wrong options in the `mrun` call
- Errors in the configuration file
- Errors in the NAMELIST parameter file
- Errors in the `ssh`-installation (authentication), if a remote host is used for batch jobs

Principal Sources of Errors

PALM runs can give rise to a large variety of errors ...

Some of the main possible reasons for errors are:

- Missing or wrong options in the `mrun` call
- Errors in the configuration file
- Errors in the NAMELIST parameter file
- Errors in the `ssh`-installation (authentication), if a remote host is used for batch jobs
- FORTRAN errors in the user code (user-interface files)

Principal Sources of Errors

PALM runs can give rise to a large variety of errors ...

Some of the main possible reasons for errors are:

- Missing or wrong options in the `mrun` call
- Errors in the configuration file
- Errors in the NAMELIST parameter file
- Errors in the `ssh`-installation (authentication), if a remote host is used for batch jobs
- FORTRAN errors in the user code (user-interface files)
- PALM runtime errors due to
  - wrong parameter settings

Principal Sources of Errors

PALM runs can give rise to a large variety of errors ...

Some of the main possible reasons for errors are:

- Missing or wrong options in the `mrun` call
- Errors in the configuration file
- Errors in the NAMELIST parameter file
- Errors in the `ssh`-installation (authentication), if a remote host is used for batch jobs
- FORTRAN errors in the user code (user-interface files)
- PALM runtime errors due to
  - wrong parameter settings
  - errors in the user code

Principal Sources of Errors

PALM runs can give rise to a large variety of errors ...

Some of the main possible reasons for errors are:

- Missing or wrong options in the `mrun` call
- Errors in the configuration file
- Errors in the NAMELIST parameter file
- Errors in the `ssh`-installation (authentication), if a remote host is used for batch jobs
- FORTRAN errors in the user code (user-interface files)
- PALM runtime errors due to
  - wrong parameter settings
  - errors in the user code
  - errors in PALM's default code, which have not been detected so far (e.g. because some parameter combinations have never been tried so far)

First Steps of Debugging

First Steps of Debugging

- **Find out the principal reason of the error(s):**

First Steps of Debugging

- **Find out the principal reason of the error(s):**
  - Carefully analyze the job protocol file (or messages on the terminal, in case of interactive runs) for any error messages or unexpected behaviour.

First Steps of Debugging

- **Find out the principal reason of the error(s):**
  - Carefully analyze the job protocol file (or messages on the terminal, in case of interactive runs) for any error messages or unexpected behaviour.
  - In case of batch runs on a remote host, if the job protocol file is missing on the local host, try if you can find it in ∼/`job_queue` on the remote host.

First Steps of Debugging

- **Find out the principal reason of the error(s):**
  - Carefully analyze the job protocol file (or messages on the terminal, in case of interactive runs) for any error messages or unexpected behaviour.
  - In case of batch runs on a remote host, if the job protocol file is missing on the local host, try if you can find it in ∼/`job_queue` on the remote host.
  - If the job has run into a time limit, no job protocol files or messages might be created at all (system depending).

First Steps of Debugging

- **Find out the principal reason of the error(s):**
  - Carefully analyze the job protocol file (or messages on the terminal, in case of interactive runs) for any error messages or unexpected behaviour.
  - In case of batch runs on a remote host, if the job protocol file is missing on the local host, try if you can find it in ∼/job_queue on the remote host.
  - If the job has run into a time limit, no job protocol files or messages might be created at all (system depending).
  - Some typical errors which may occur during execution of mrun are automatically detected and displayed by mrun in the job protocol or on the terminal:

    Respective error messages will begin with "+++".

First Steps of Debugging

- **Find out the principal reason of the error(s):**
  - Carefully analyze the job protocol file (or messages on the terminal, in case of interactive runs) for any error messages or unexpected behaviour.
  - In case of batch runs on a remote host, if the job protocol file is missing on the local host, try if you can find it in ∼/job_queue on the remote host.
  - If the job has run into a time limit, no job protocol files or messages might be created at all (system depending).
  - Some typical errors which may occur during execution of mrun are automatically detected and displayed by mrun in the job protocol or on the terminal:

    Respective error messages will begin with "+++".
  - Compile and runtime error messages will only appear in the job protocol or on the terminal (in case of interactive runs).

First Steps of Debugging

- **Find out the principal reason of the error(s):**
  - Carefully analyze the job protocol file (or messages on the terminal, in case of interactive runs) for any error messages or unexpected behaviour.
  - In case of batch runs on a remote host, if the job protocol file is missing on the local host, try if you can find it in ∼/job_queue on the remote host.
  - If the job has run into a time limit, no job protocol files or messages might be created at all (system depending).
  - Some typical errors which may occur during execution of mrun are automatically detected and displayed by mrun in the job protocol or on the terminal:

    Respective error messages will begin with "+++".
  - Compile and runtime error messages will only appear in the job protocol or on the terminal (in case of interactive runs).
  - In case of runtime errors terminal messages may give first helpful hints about where errors are located.

Debugging Runtime Errors (I)

- In case of runtime errors, the available information depends on the compiler and on the compiler settings.

Debugging Runtime Errors (I)

- In case of runtime errors, the available information depends on the compiler and on the compiler settings.
- The default options for the Intel-compiler (-O3 for fast execution) give almost no information, e.g. about the subroutine or the line number of the code where the error occured. Execution is even continued in case of floating point errors!

Debugging Runtime Errors (I)

- In case of runtime errors, the available information depends on the compiler and on the compiler settings.
- The default options for the Intel-compiler (-O3 for fast execution) give almost no information, e.g. about the subroutine or the line number of the code where the error occured. Execution is even continued in case of floating point errors!
- Floating point error detection and traceback can be activated with compiler options

```
ifort -fpe0 -debug -traceback -O0 ...
```

Debugging Runtime Errors (I)

- In case of runtime errors, the available information depends on the compiler and on the compiler settings.
- The default options for the Intel-compiler (-O3 for fast execution) give almost no information, e.g. about the subroutine or the line number of the code where the error occured. Execution is even continued in case of floating point errors!
- Floating point error detection and traceback can be activated with compiler options

```
ifort -fpe0 -debug -traceback -O0 ...
```

- The default `.mrun.config.default` file contains an additional block with debug options. It can be used with mrun-call

```
mrun ...  -h <host_identifier> -K "parallel trace"...
```

Debugging Runtime Errors (II)

- The configuration file `.mrun.config.default` looks like this:

Debugging Runtime Errors (II)

- The configuration file `.mrun.config.default` looks like this:

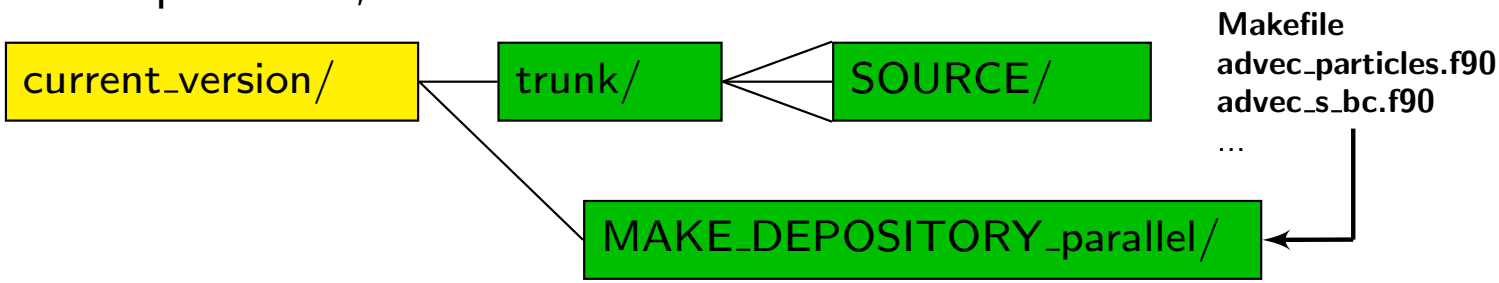
```
.
# The next line is just an example. Add your own line below or replace this line.
%host_identifier   myhostname  lcmy
#
# The next block contains all informations for compiling the PALM code
# This is the block for the optimized version
%fopts             -I:<replace by mpi include path>:-fpe0:-O3:-xHost:...    <hi> parallel
%lopts             -L:<replace by mpi library path>:-fpe0:-O3:-xHost:...    <hi> parallel
.
.
# This is the block for the debug version
%fopts             -C:-check:nooutput_conversion:-fpe0:-debug:...           <hi> parallel trace
%lopts             -C:-check:nooutput_conversion:-fpe0:-debug:...           <hi> parallel trace
.
```

Debugging Runtime Errors (III)

- If you now call mbuild, it will first compile for the optimized version, and then for the debug version. The pre-compiled code will be put into different make depositories, one for each block:

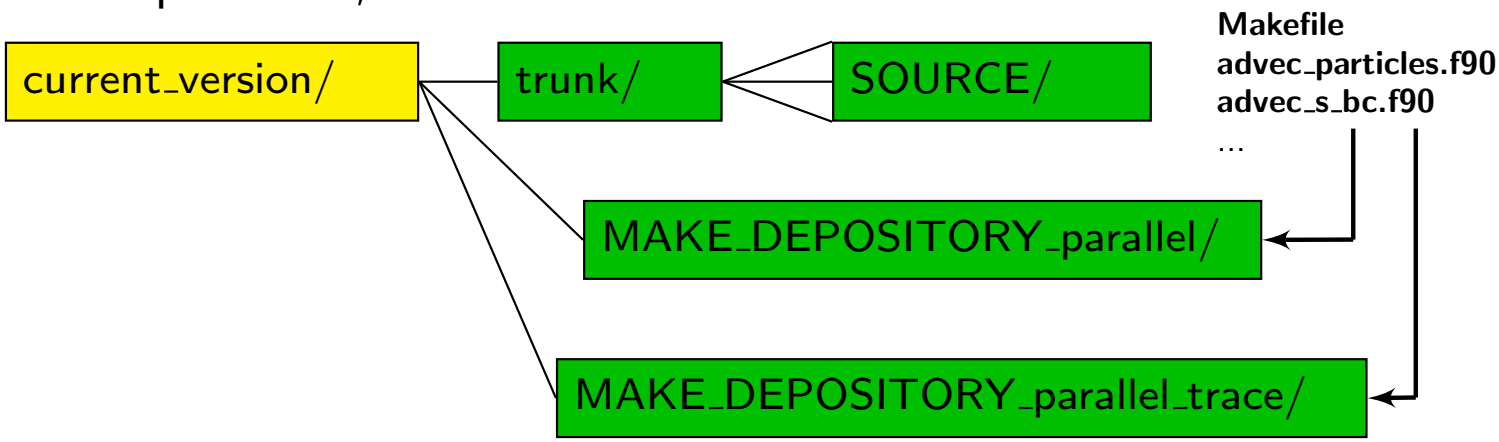
Debugging Runtime Errors (III)

- If you now call mbuild, it will first compile for the optimized version, and then for the debug version. The pre-compiled code will be put into different make depositories, one for each block:



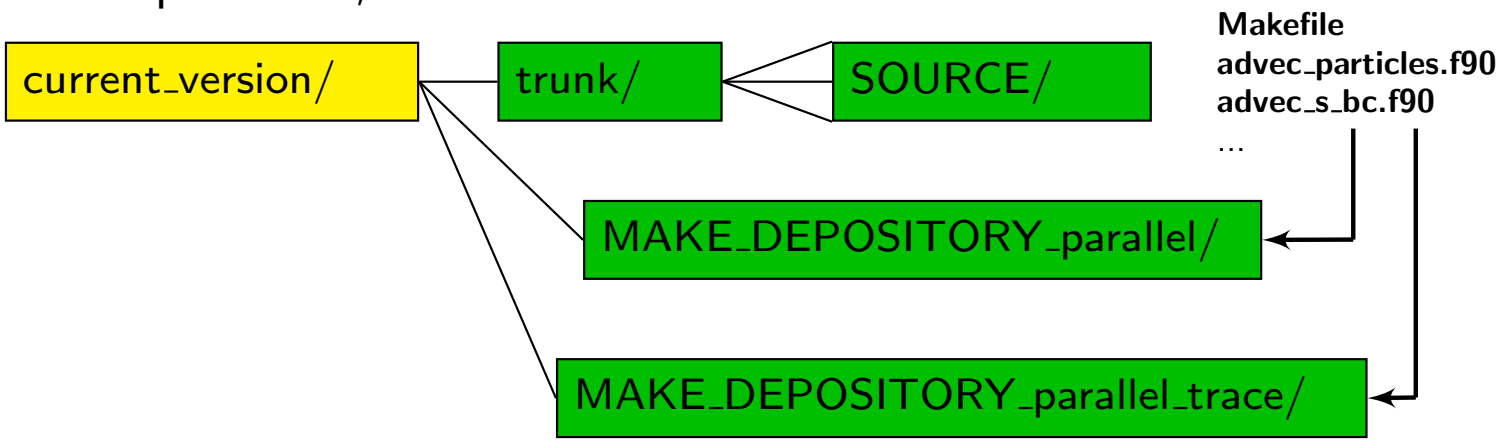
Debugging Runtime Errors (III)

- If you now call mbuild, it will first compile for the optimized version, and then for the debug version. The pre-compiled code will be put into different make depositories, one for each block:



Debugging Runtime Errors (III)

- If you now call mbuild, it will first compile for the optimized version, and then for the debug version. The pre-compiled code will be put into different make depositories, one for each block:

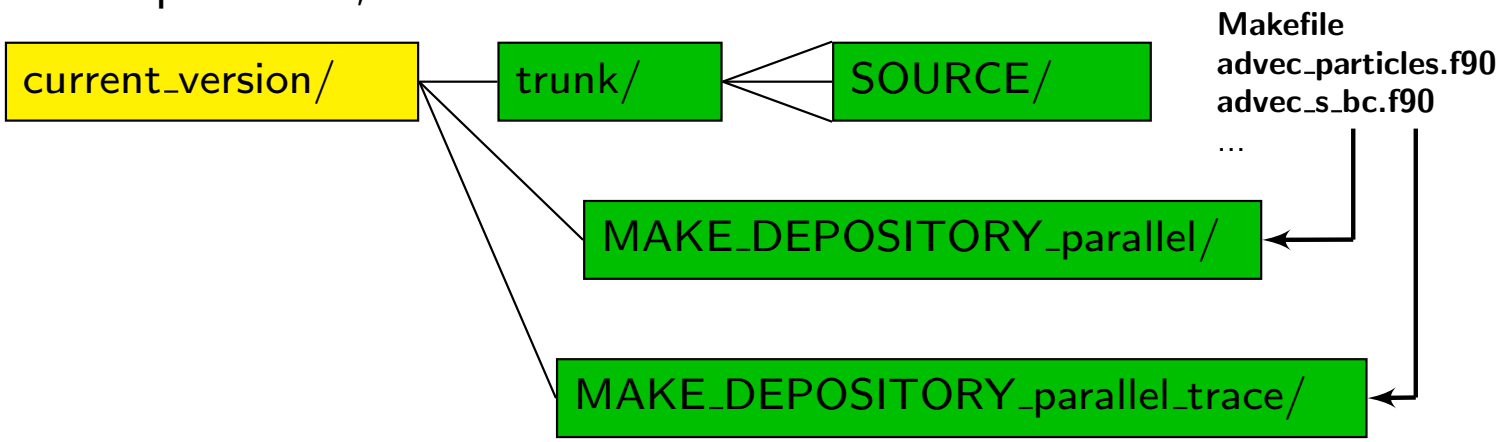


- The `mrun` option `-K` defines, which version is used:

`mrun ... -K parallel ...` will use the optimized version
`mrun ... -K "parallel trace"...` will use the debug version

Debugging Runtime Errors (III)

- If you now call mbuild, it will first compile for the optimized version, and then for the debug version. The pre-compiled code will be put into different make depositories, one for each block:



- The `mrun` option `-K` defines, which version is used:

`mrun ... -K parallel ...` will use the optimized version
`mrun ... -K "parallel trace"...` will use the debug version

Enabling debug options slows down the execution speed significantly!

Debugging Runtime Errors (IV)

Debugging Runtime Errors (IV)

- There are still some cases, where these options do not help or do not give enough information (e.g. concerning segmentations faults).

Debugging Runtime Errors (IV)

- There are still some cases, where these options do not help or do not give enough information (e.g. concerning segmentations faults).
- There are two ways of handling these cases:

Debugging Runtime Errors (IV)

- There are still some cases, where these options do not help or do not give enough information (e.g. concerning segmentations faults).
- There are two ways of handling these cases:
  - the print/write debugger

Debugging Runtime Errors (IV)

- There are still some cases, where these options do not help or do not give enough information (e.g. concerning segmentations faults).
- There are two ways of handling these cases:
  - the print/write debugger
  - debuggers like dbx or GUI-based debuggers like “totalview“ or “Allinea DDT“

Debugging Runtime Errors (IV)

- There are still some cases, where these options do not help or do not give enough information (e.g. concerning segmentations faults).
- There are two ways of handling these cases:
  - the print/write debugger
  - debuggers like dbx or GUI-based debuggers like “totalview“ or “Allinea DDT“
  - more detailed informations about using Allinea DDT on the HLRN-III system are given under:

    https://www.hlrn.de/home/view/System3/AllineaDDT

Debugging Runtime Errors (IV)

- There are still some cases, where these options do not help or do not give enough information (e.g. concerning segmentations faults).
- There are two ways of handling these cases:
  - the print/write debugger
  - debuggers like dbx or GUI-based debuggers like “totalview“ or “Allinea DDT“
  - more detailed informations about using Allinea DDT on the HLRN-III system are given under:

    https://www.hlrn.de/home/view/System3/AllineaDDT

  - `mrun`-script will soon be adjusted for allowing to use the “Allinea DDT“ debugger.

Debugging With Print Statements (I)

- By adding appropriate print statements to the code

```
WRITE(9,*) 'now i am at #1'
CALL local_flush( 9 )
WRITE(9,*) 'now i am at #2'
CALL local_flush( 9 )
...
```

  you can find the exact position (line number) within the code, where the error occurs.

Debugging With Print Statements (I)

- By adding appropriate print statements to the code

```
WRITE(9,*) 'now i am at #1'
CALL local_flush( 9 )
WRITE(9,*) 'now i am at #2'
CALL local_flush( 9 )
...
```

  you can find the exact position (line number) within the code, where the error occurs.

- Output can be found in files `DEBUG_0000`, `DEBUG_0001`, etc. in PALM's temporary working directory. You have to keep this directory using `mrun`-option "`-B`", because otherwise, the temporary working directory is deleted at the end of the run!

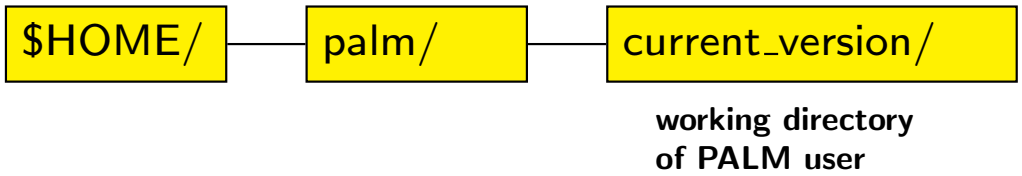
Debugging With Print Statements (I)

- By adding appropriate print statements to the code

```
WRITE(9,*) 'now i am at #1'
CALL local_flush( 9 )
WRITE(9,*) 'now i am at #2'
CALL local_flush( 9 )
...
```

  you can find the exact position (line number) within the code, where the error occurs.

- Output can be found in files `DEBUG_0000`, `DEBUG_0001`, etc. in PALM's temporary working directory. You have to keep this directory using `mrun`-option "`-B`", because otherwise, the temporary working directory is deleted at the end of the run!



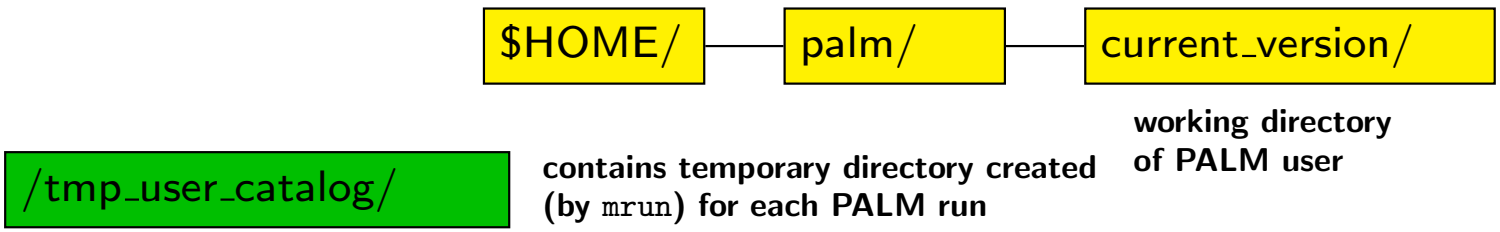
Debugging With Print Statements (I)

- By adding appropriate print statements to the code

```
WRITE(9,*) 'now i am at #1'
CALL local_flush( 9 )
WRITE(9,*) 'now i am at #2'
CALL local_flush( 9 )
...
```

  you can find the exact position (line number) within the code, where the error occurs.

- Output can be found in files `DEBUG_0000`, `DEBUG_0001`, etc. in PALM's temporary working directory. You have to keep this directory using `mrun`-option "`-B`", because otherwise, the temporary working directory is deleted at the end of the run!



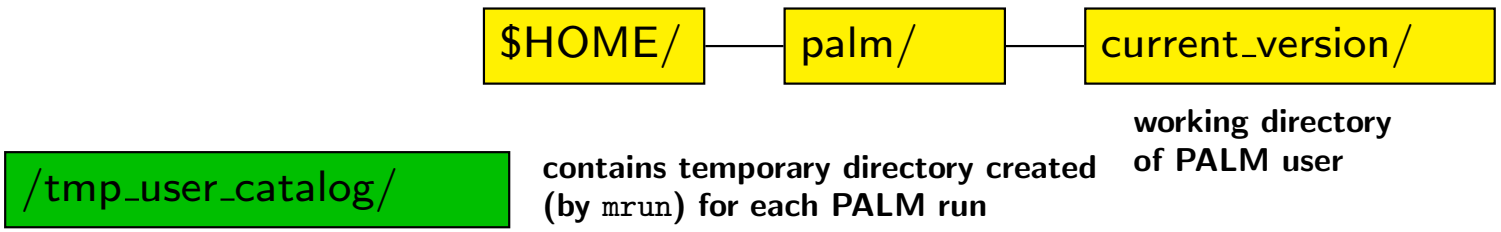
Debugging With Print Statements (I)

- By adding appropriate print statements to the code

```
WRITE(9,*) 'now i am at #1'
CALL local_flush( 9 )
WRITE(9,*) 'now i am at #2'
CALL local_flush( 9 )
...
```

  you can find the exact position (line number) within the code, where the error occurs.
- Output can be found in files `DEBUG_0000`, `DEBUG_0001`, etc. in PALM's temporary working directory. You have to keep this directory using `mrun`-option "`-B`", because otherwise, the temporary working directory is deleted at the end of the run!
- The name of PALM's temporary working directory is generated from environment variable `tmp_user_catalog` (see `.mrun.config`), the username, and a random number:

`/<tmp_user_catalog>/<username>.<random number>`



Debugging With Print Statements (II)

Debugging With Print Statements (II)

- After having located the position, you can try to find out which variable may have caused the error:

```
WRITE(9,*) ' a=',a,' b=',b, ...
```

Debugging With Print Statements (II)

- After having located the position, you can try to find out which variable may have caused the error:

```
WRITE(9,*) ' a=',a,' b=',b, ...
```

Very important: Every output is buffered, i.e. it will not be directly written on disc. In case of program aborts, the buffer contents are lost, so the output of the last write statements are not available. You have to prevent this problem by flushing the buffer after each print/write statement:

```
WRITE(9,*) '...'
CALL local_flush( 9 )
```