ami_bitstructs.py

```
from construct import Struct, BitStruct, BitsInteger, FormatField, BitsSwapped
from .revbitinteger import RevBitsInteger
from .consts import (
    DATEPACKED,
    DAY,
    MONTH,
    YEAR,
    VOLUME,
    CLOSE,
     OPEN,
     HIGH,
     LOW,
     FUT,
     RESERVED,
     MICRO_SEC,
     MILLI_SEC,
     SECOND,
     MINUTE,
     HOUR,
     AUX_1,
     AUX_2,
     TERMINATOR,
)


SwappedField = BitsSwapped(FormatField("<", "f"))

DateShort=BitsSwapped(BitStruct(
    MINUTE / BitsInteger(length=6),  # 38
    HOUR / BitsInteger(length=5),  # 43
    DAY / RevBitsInteger(length=5),  # Bit 48 Byte 6
    MONTH / RevBitsInteger(length=4),  # 52
    YEAR / RevBitsInteger(length=12),
))

Date=BitStruct(
    FUT / BitsInteger(length=1),  # 1
    RESERVED / BitsInteger(length=5),  # 6
    MICRO_SEC / BitsInteger(length=10),  # Bit 16 byte 2
    MILLI_SEC / BitsInteger(length=10),  # 26
    SECOND / BitsInteger(length=6),  # Bit 32 Byte 4
    MINUTE / BitsInteger(length=6),  # 38
    HOUR / BitsInteger(length=5),  # 43
    DAY / RevBitsInteger(length=5),  # Bit 48 Byte 6
    MONTH / RevBitsInteger(length=4),  # 52
    YEAR / RevBitsInteger(length=12),  # Bit  64 Byte 8
)
EntryChunk = Struct(
    DATEPACKED
    / Date,
    CLOSE / SwappedField, # Byte 4
    OPEN / SwappedField,
    HIGH / SwappedField,
    LOW / SwappedField,
    VOLUME / SwappedField,  # 160
    AUX_1 / SwappedField,
    AUX_2 / SwappedField,
    TERMINATOR / SwappedField,  # 256 + 64
)


def create_entry_chunk():
    return EntryChunk

```

ami_construct.py

```
from construct import (
    Struct,
    Bytes,
    GreedyRange,
    PaddedString,
    swapbitsinbytes,
    BitsSwapped,
    bytes2bits,
    bits2bytes,
     Const,
     CString,
     Padded,
     BitsInteger,
     Int32ul,
     FormatField
)
from .consts import (
     DATEPACKED,
     DAY,
     MONTH,
     YEAR,
     VOLUME,
     CLOSE,
     OPEN,
     HIGH,
     LOW,
     FUT,
     RESERVED,
     MICRO_SEC,
     MILLI_SEC,
     SECOND,
     MINUTE,
     HOUR,
     AUX_1,
     AUX_2,
     TERMINATOR,
)
from .ami_bitstructs import EntryChunk,DateShort
import struct

DIVIDEND_PAY_DATE = "Dividend Pay Date"

DELISTING_DATE = "Delisting Date"
SwappedField = FormatField("<", "f")

from construct import CString

ascii_str = CString("ascii")
Master = Struct(
    "Header" / Bytes(8),
    "NumSymbols" / Int32ul,
    "Symbols"
    / GreedyRange(
        Struct(
            "Symbol" / Padded(492, CString("ascii")),
            "CONST"
            / Const(
                b"\x02\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x80\x3F"
```

```
            ),
            "Rest" / Bytes(1172 - 5 - 16 - 490 + 3),
        )
    ),
)

SymbolHeader = Struct(
    "Start" / Const(b"BROKDAt5"),# 8
    "SymbolName" / Padded(144, CString("ascii")), #144 + 8 = 152
    "FullName" / Padded(348, CString("ascii")), # 152+349 =501
    "First_Const"/SwappedField,
    "Second_Const" / SwappedField,
    "Third_Const" / SwappedField,
    "Fourth_Const" / SwappedField,
    "Round Lot Size" / SwappedField,
    "Tick size" / SwappedField,
    "Filler_3" / SwappedField,
    "Filler_4" / SwappedField,
    "Filler_5" / SwappedField,
    "Last Split Date" / DateShort,
    "Filler_7" / SwappedField,
    DIVIDEND_PAY_DATE / DateShort,
    "Filler_9" / SwappedField,
    "Ex-Dividend Date" / DateShort,
    "Shares Float" / SwappedField,
    "Shares Out" / SwappedField,
    "Dividend" / SwappedField,
    "Book Value(p.s.)" / SwappedField,
    "PEG Ratio" / SwappedField,
    "Profit Margin" / SwappedField,
    "Operating Margin" / SwappedField,
    "1yr target price" / SwappedField,
    "Return on Assets (ttm)"/SwappedField,
    "Return on Equity (ttm)"/SwappedField,
    "Qtrly Rev. Growth"/SwappedField,
    "Gross Profit (p.s)"/SwappedField,
     "Sales Per Share"/SwappedField,
     "EBITDA (p.s)"/SwappedField,
     "Qtrly Earnings Growth"/SwappedField,
     "% Held by Insiders"/SwappedField,
     "% Held by Institutions"/SwappedField,
     "Shares Short"/SwappedField,
     "Shares Short Prior Month"/SwappedField,
     "Forward EPS"/SwappedField,
     "EPS"/SwappedField,
     "EPS Est. Current Year"/SwappedField,
     "EPS Est. Next Year"/SwappedField,
     "EPS Est. Next Quarter"/SwappedField,
     "Forward Dividend"/SwappedField,
     "Beta"/SwappedField,
     "Operating Cash Flow"/SwappedField,
     "Levered Free Cash Flow"/SwappedField,
     "Filler_39"/SwappedField,
     "Filler_40"/SwappedField,
     "Filler_41"/SwappedField,
     "Filler_42"/SwappedField, # 685
     "A Date"/SwappedField,
     "Filler_44"/SwappedField,
     DELISTING_DATE/DateShort,
     "Filler_46"/SwappedField,
     "Filler_47"/SwappedField,
     "Filler_48"/SwappedField,
     "Filler_49"/SwappedField,
     "Filler_50"/SwappedField,
     "Filler_51"/SwappedField,
     "Filler_52"/SwappedField,
     "Filler_53"/SwappedField,
     "Filler_54"/SwappedField,
     "Filler_55"/SwappedField,
     "Filler_56"/SwappedField,
     "Filler_57"/SwappedField,
     "Filler_58"/SwappedField,
     "Filler_59"/SwappedField,
     "Filler_60"/SwappedField,
     "Filler_61"/SwappedField,
     "Filler_62"/SwappedField,
     "Filler_63"/SwappedField,
     "Filler_64"/SwappedField,
     "Filler_65"/SwappedField,
     "Filler_66"/SwappedField,
     "Filler_67"/SwappedField,
     "Space"/Bytes(396),
     "Length"/Int32ul
)
SymbolConstruct = Struct(
     "Header" / Bytes(0x4A0), "Entries" / GreedyRange(BitsSwapped(EntryChunk))
)

```

ami_database.py

```
from .ami_reader import AmiReader
from .ami_dataclasses import SymbolEntry, SymbolData
from .ami_construct import Master, SymbolConstruct
from pathlib import Path
import os

from .ami_database_folder_layout import AmiDbFolderLayout


def symbolpath(root, symbol):
    return os.path.join(root, f"{symbol[0].lower()}/{symbol}")


class AmiDataBase(AmiDbFolderLayout):
    def __init__(self, folder, use_compiled=False, avoid_windows_file=True):
        if not os.path.exists(folder):
            os.mkdir(folder)
        self.avoid_windows_file = avoid_windows_file
        self.reader = AmiReader(folder, use_compiled=use_compiled)
        self._symbol_cache = {}
        self._fast_symbol_cache = {}
        self._symbols = []
        self._symbol_frames = {}
        self._modified_symbols = []
        self._master = self.reader.get_master()
        self.folder = folder
        self._master_path = os.path.join(folder, "broker.master")

    def get_symbols(self):
        if len(self._symbols) == 0:
            self._symbols = self.reader.get_symbols()
        return self._symbols

    def add_symbol(self, symbol_name):
        self._master.append_symbol(symbol=symbol_name)

    def add_new_symbol(self, symbol_name, symboldata=None):
        if self.avoid_windows_file:
```

```
            new_symbol_name = self._replace_windows_pipe_file(symbol_name)
            self._add_new_symbol(new_symbol_name, symboldata)
        else:
            self._add_new_symbol(symbol_name, symboldata)

    def _replace_windows_pipe_file(self, symbol_name):
        wfiles = ["CON", "AUX", "LST", "PRN", "NUL", "EOF", "INP", "OUT"]
        result = symbol_name
        if symbol_name[:3] in wfiles:
            result = symbol_name.replace(symbol_name[:3], "_".join(symbol_name[:3]))
        return result

    def _add_new_symbol(self, symbol_name, symboldata=None):
        self._master.append_symbol(symbol=symbol_name)
        self.read_fast_data_for_symbol(symbol_name)
        if isinstance(symboldata, dict):
            self._fast_symbol_cache[symbol_name] += symboldata
        if isinstance(symboldata, list):
            for el in symboldata:
                self._fast_symbol_cache[symbol_name] += el

    def append_to_symbol(self, symbol_name, symboldata):
        if symbol_name not in self._fast_symbol_cache:
            self.read_fast_data_for_symbol(symbol_name)
        if type(symboldata) == dict:
            self._fast_symbol_cache[symbol_name] += symboldata
        if type(symboldata) == list:
            for el in symboldata:
                self._fast_symbol_cache[symbol_name] += el

    def add_symbol_data_dict(self, input_dict):
        pass

    def store_symbol(self, symbol_name):
        if symbol_name in self._symbol_cache:
            data = self._symbol_cache[symbol_name].to_construct_dict()
            newbin = SymbolConstruct.build(data)
            f = open(os.path.join(self.folder, symbol_name), "wb")
            try:
                f.write(newbin)
            finally:
                f.close()

    def ensure_symbol_folder(self, symbol):
        symb_root=self.get_symbol_root_folder(symbol)
        Path(os.path.join(self.folder, symb_root)).mkdir(
            parents=True, exist_ok=True
        )

    def write_database(self):
        con_data = self._master.to_construct_dict()
        newbin = Master.build(con_data)
        f = open(self._master_path, "wb")
        try:
            f.write(newbin)
        finally:
            f.close()
        for symbol in self._fast_symbol_cache:
            newbin = self._fast_symbol_cache[symbol].binary
            self.ensure_symbol_folder(symbol)
            f = open(self._get_symbol_path(self.folder, symbol), "wb")
             try:
                 f.write(newbin)
             finally:
                 f.close()

        for symbol in self._symbol_cache:
            newbin = SymbolConstruct.build(
                self._symbol_cache[symbol].to_construct_dict()
            )
            self.ensure_symbol_folder(symbol)
            f = open(self._get_symbol_path(self.folder, symbol), "wb")
            try:
                f.write(newbin)
            finally:
                f.close()

    def read_data_for_symbol(self, symbol_name):
        self._symbol_cache[symbol_name] = self.reader.get_symbol_data(symbol_name)

    def read_fast_data_for_symbol(self, symbol_name):
        self._fast_symbol_cache[symbol_name] = self.reader.get_fast_symbol_data(
            symbol_name
        )

    def read_raw_data_for_symbol(self, symbol_name):
        return self.reader.get_symbol_data_raw(symbol_name)

    def get_dict_for_symbol(self, symbol_name):
        if symbol_name in self._symbol_cache:
            return self._symbol_cache[symbol_name].to_dict()

        self.read_data_for_symbol(symbol_name)
        return self._symbol_cache[symbol_name].to_dict()

    def get_symbol_data(self, symbol_name):
        if symbol_name in self._symbol_cache:
            return self._symbol_cache[symbol_name]

        self.read_data_for_symbol(symbol_name)
        return self._symbol_cache[symbol_name]

    def get_fast_symbol_data(self, symbol_name):
        if symbol_name in self._fast_symbol_cache:
            return self._fast_symbol_cache[symbol_name]

        self.read_fast_data_for_symbol(symbol_name)
        return self._fast_symbol_cache[symbol_name]

    def append_symbole_entry(self, symbol, data: SymbolEntry):
        """
        :param symbol: name of the symbol to which data should be appended
        :param data: Instance of SymbolEntry
        :return:
        """
        self._modified_symbols.append(symbol)
        if symbol not in self._symbol_cache:
            self._symbol_cache[symbol] = SymbolData(Entries=[data])
            return

        self._symbol_cache[symbol].append(data)

    def append_symbol_data(self, symbol_data):
        """

        :param symbol_data:
        :return:
        """

        assert type(symbol_data) == dict
```

```
        for symbol in symbol_data:
            assert type(symbol_data[symbol]) == dict
            all(el in symbol_data[symbol] for el in SymbolEntry.get_necessary_args())
            symbol_lengths = [len(symbol_data[symbol][k]) for k in symbol_data[symbol]]
            assert min(symbol_lengths) == max(symbol_lengths)
            data = [
                SymbolEntry(
                    **{k: symbol_data[symbol][k][i] for k in symbol_data[symbol]}
                )
                for i in range(max(symbol_lengths))
            ]
            if symbol not in self._symbol_cache:
                self._symbol_cache[symbol] = SymbolData(Entries=data)
            else:
                self._symbol_cache[symbol].append(data)

```

ami_database_folder_layout.py

```
import os

ERROR_RETURNED = True


class AmiDbFolderLayout:
    def get_symbol_root_folder(self, symbol):
        if symbol[0] in ["^", "~","@"]:
            return "_"
        return symbol[0].lower()

    def _get_symbol_path(self, root, symbol):
        assert os.path.isdir(root), f"{root} is not a directory"
        if symbol.lower() == "broker.master":
            return os.path.join(root, f"{symbol}")

        symbol_folder = self.get_symbol_root_folder(symbol)
        return os.path.join(root, f"{symbol_folder}/{symbol}")

```

ami_dataclasses.py

```
from dataclasses import dataclass, field, fields
from dataclass_type_validator import dataclass_validate
from typing import List
from .consts import (
    DAY,
    MONTH,
    CLOSE,
    OPEN,
    HIGH,
     LOW,
     VOLUME,
     YEAR,
     DATEPACKED,
     FUT,
     RESERVED,
     MICRO_SEC,
     AUX_1,
     AUX_2,
     TERMINATOR,
     MILLI_SEC,
     SECOND,
     MINUTE,
     HOUR,
)
from .ami_construct import SymbolConstruct, Master

SYMBOL_REST = b"\0" * (1172 - 5 - 16 - 490 + 3)
SYMBOL_SPACE = b"\0" * (495 - 5 - 3)
SYMBOL_STR = b"\0" * (497)


@dataclass()
class SymbolEntry:
    Month: int = 0
    Year: int = 0
    Close: float = 0.0
    Open: float = 0.0
    Low: float = 0.0
    High: float = 0.0
    Volume: float = 0.0
    Future: int = 0
    Reserved: int = 0
    Micro_second: int = 0
    Milli_sec: int = 0
    Second: int = 0
    Minute: int = 0
    Hour: int = 0
    Day: int = 0
    Aux_1: int = 0
    Aux_2: int = 0
    Terminator: int = 0

    def __post_init__(self):
        current_fields = fields(self)
        for field in current_fields:
            field_name = field.name
            expected_type = field.type
            value = getattr(self, field_name)
            if expected_type == int:
                assert isinstance(value, int)

            if expected_type == float:
                if isinstance(value, int):
                    self.__setattr__(field_name, float(value))
                value = getattr(self, field_name)
                assert isinstance(value, float)

    @classmethod
    def get_necessary_args(self):
        return ["Month", "Year", "Day", "Close", "High", "Open", "Low", "Volume"]

    def set_by_construct(self, con_data):
        date_data = con_data[DATEPACKED]
        self.Future = date_data[FUT]
        self.Reserved = date_data[RESERVED]
        self.Micro_second = date_data[MICRO_SEC]
        self.Milli_sec = date_data[MILLI_SEC]
        self.Second = date_data[SECOND]
        self.Minute = date_data[MINUTE]
        self.Hour = date_data[HOUR]
        self.Day = date_data[DAY]
        self.Month = date_data[MONTH]
        self.Year = date_data[YEAR]

        self.Close = con_data[CLOSE]
        self.Open = con_data[OPEN]
        self.High = con_data[HIGH]
```

```
        self.Low = con_data[LOW]
        self.Volume = con_data[VOLUME]
        self.Aux_1 = con_data[AUX_1]
        self.Aux_2 = con_data[AUX_2]
        self.Terminator = con_data[TERMINATOR]
        return self

    def to_construct_dict(self):
        return {
            DATEPACKED: {
                FUT: self.Future,
                 RESERVED: self.Reserved,
                 MICRO_SEC: self.Micro_second,
                 MILLI_SEC: self.Milli_sec,
                 SECOND: self.Second,
                 MINUTE: self.Minute,
                 HOUR: self.Hour,
                 DAY: self.Day,
                 MONTH: self.Month,
                 YEAR: self.Year,
            },
            CLOSE: self.Close,
            OPEN: self.Open,
            HIGH: self.High,
            LOW: self.Low,
            VOLUME: self.Volume,  # 160
            AUX_1: self.Aux_1,
            AUX_2: self.Aux_2,
            TERMINATOR: self.Terminator,
        }

        pass


@dataclass_validate()
@dataclass()
class SymbolData:
    Header: bytes = b"\0" * 0x4A0
    Entries: List[SymbolEntry] = field(default_factory=list)

    def append(self, entry: SymbolEntry):
        self.Entries.append(entry)

    def set_by_construct(self, con_data):
        self.Header = con_data["Header"]
        self.Entries = [
            SymbolEntry().set_by_construct(el) for el in con_data["Entries"]
        ]
        return self

    def to_dict(self):
        result = {
            DAY: [],
            MONTH: [],
            YEAR: [],
            OPEN: [],
            HIGH: [],
            LOW: [],
            CLOSE: [],
            VOLUME: [],
        }
        for el in self.Entries:
            result[DAY].append(el.Day)
            result[MONTH].append(el.Month)
            result[YEAR].append(el.Year)

            result[OPEN].append(el.Open)
            result[HIGH].append(el.High)
            result[LOW].append(el.Low)
            result[CLOSE].append(el.Close)
            result[VOLUME].append(el.Volume)
        return result

    def to_construct_dict(self):
        result = {
            "Header": self.Header,
            "Entries": [el.to_construct_dict() for el in self.Entries],
        }
        return result

    def write_to_file(self, file):
        binary = SymbolConstruct.build(self.to_construct_dict())
        file.write(binary)


@dataclass_validate()
@dataclass()
class MasterEntry:
    Symbol: str = ""
    Rest: bytes = SYMBOL_REST

    def to_construct_dict(self):
        result = {"Symbol": self.Symbol, "Rest": self.Rest, "Const": None}
        return result

    def set_by_construct(self, con_data):
        if type(con_data["Symbol"]) != str:
            return self
        self.Symbol = con_data["Symbol"]
        self.Rest = con_data["Rest"]
        return self


@dataclass_validate()
@dataclass()
class MasterData:
    Header: bytes = b"BROKMAS2"
    NumSymbols: int = 0
    Symbols: List[MasterEntry] = field(default_factory=list)

    def write_to_file(self, file):
        Master.build()

    def append_symbol(self, symbol: str, rest: bytes = SYMBOL_REST):
        self.Symbols.append(MasterEntry(Symbol=symbol, Rest=rest))
        self.NumSymbols= len(self.Symbols)


    def get_symbols(self):
        return [el.Symbol for el in self.Symbols]

    def to_construct_dict(self):
        result = {
            "Header": self.Header,
            "NumSymbols": self.NumSymbols,
            "Symbols": [el.to_construct_dict() for el in self.Symbols],
        }
        return result

    def set_by_construct(self, con_data):
```

```
        self.Header = con_data["Header"]
        self.NumSymbols = con_data["NumSymbols"]
        self.Symbols = [
            MasterEntry().set_by_construct(el) for el in con_data["Symbols"]
        ]
        return self

```

ami_reader.py

```
from construct import Struct, Bytes, GreedyRange
from .ami_dataclasses import SymbolEntry, SymbolData, MasterData
from .ami_construct import Master, SymbolConstruct
from .ami_symbol_facade import AmiSymbolDataFacade
from .consts import YEAR, DAY, MONTH, CLOSE, OPEN, HIGH, LOW, VOLUME, DATEPACKED
import os
from .ami_database_folder_layout import AmiDbFolderLayout

ERROR_RETURNED = True

VALUE_INDEX = 2
BROKER_MASTER = "broker.master"


class AmiReader(AmiDbFolderLayout):
    def __init__(self, folder, use_compiled=False):
        self.__folder = folder
        self.__symbol = SymbolConstruct
        self.__master = Master
        if use_compiled:
            self.__symbol = SymbolConstruct.compile(
                filename=os.path.join(os.path.dirname(__file__), "SymbolConstruct.py")
            )
            self.__master = Master.compile(
                filename=os.path.join(os.path.dirname(__file__), "Master.py")
            )

        self.__master = self._read_master()
        self.__symbols = self.__read_symbols()

    def get_master(self):
        return self.__master

    def _read_master(self):
        binarry, errorstate, errmsg = self.__get_binarry(BROKER_MASTER)
        if errorstate:
            return MasterData()

        parsed = Master.parse(binarry)
        return MasterData().set_by_construct(parsed)

    def __read_symbols(self):
        return self.__master.get_symbols()

    def __get_binarry(self, symbol_name):
        """

        :param filename:
        :return: binarray, error state, errormsg
        """
        filename=self._get_symbol_path(self.__folder, symbol_name)
        if not os.path.isfile(filename):
            return [], ERROR_RETURNED, f"{filename} is not a file"
        binarry = open(filename, "rb").read()
        return binarry, False, ""

    def get_symbols(self):
        return self.__symbols.copy()

    def get_fast_symbol_data(self, symbol_name):
        binarry, errorstate, errmsg = self.__get_binarry(
            symbol_name
        )
        if errorstate:
            return AmiSymbolDataFacade()
        return AmiSymbolDataFacade(binarry)

    def get_symbol_data_raw(self, symbol_name):
        binarry, errorstate, errmsg = self.__get_binarry(symbol_name)
        if errorstate:
            return []
        data = self.__symbol.parse(binarry)
        return data

    def get_symbol_data_dictionary(self, symbol_name):
        symbdata = self.get_symbol_data_raw(symbol_name)
        if type(symbdata) == dict:
            return {}
        packed_map = {
            DAY: lambda x: x[DATEPACKED][DAY],
            MONTH: lambda x: x[DATEPACKED][MONTH],
            YEAR: lambda x: x[DATEPACKED][YEAR],
        }
        data_lines = symbdata["Entries"]
        result = {
            DAY: [],
            MONTH: [],
            YEAR: [],
            OPEN: [],
            HIGH: [],
            LOW: [],
            CLOSE: [],
            VOLUME: [],
        }
        for el in data_lines:
            for k in result:
                if k in [DAY, MONTH, YEAR]:
                    result[k].append(el[DATEPACKED][k])
                else:
                    result[k].append(el[k])

        return result

    def get_symbol_data(self, symbol_name):
        binarry, errorstate, errmsg = self.__get_binarry(symbol_name)
        if errorstate == ERROR_RETURNED:
            return SymbolData()

        data = self.__symbol.parse(binarry)
        values = [
            SymbolEntry(
                Open=el[OPEN],
                Low=el[LOW],
                High=el[HIGH],
                Close=el[CLOSE],
                Volume=el[VOLUME],
                Day=el[DATEPACKED][DAY],
                Month=el[DATEPACKED][MONTH],
                Year=el[DATEPACKED][YEAR],
```

```
            )
            for el in data["Entries"]
        ]
        return SymbolData(Header=data["Header"], Entries=values)

```

ami_symbol_facade.py

```
from construct import (
    Struct,
    Bytes,
    GreedyRange,
    PaddedString,
    swapbitsinbytes,
    BitsSwapped,
    bytes2bits,
    bits2bytes,
)
from .consts import (
    DATEPACKED,
    DAY,
    MONTH,
    YEAR,
    VOLUME,
    CLOSE,
    OPEN,
    HIGH,
    LOW,
    FUT,
    RESERVED,
    MICRO_SEC,
    MILLI_SEC,
    SECOND,
    MINUTE,
    HOUR,
    AUX_1,
    AUX_2,
    TERMINATOR,
)
from .ami_bitstructs import EntryChunk
from .ami_construct import SymbolHeader
import struct

entry_map = [
    DAY,
    MONTH,
    YEAR,
    MICRO_SEC,
    MILLI_SEC,
    SECOND,
    MINUTE,
    HOUR,
    VOLUME,
    AUX_1,
    AUX_2,
    TERMINATOR,
    CLOSE,
    OPEN,
    HIGH,
    LOW,
    FUT,
]

NUM_HEADER_BYTES = 0x4A0

OVERALL_ENTRY_BYTES = 40

TERMINATOR_DOUBLE_WORD_LENGTH = 4

Master = Struct(
    "Header" / Bytes(0x4A0),
    "Symbols"
    / GreedyRange(
        Struct("Symbol" / PaddedString(5, "ASCII"), "Rest" / Bytes(1172 - 5))
    ),
)


SymbolConstruct = Struct(
    "Header" / Bytes(0x4A0), "Entries" / GreedyRange(BitsSwapped(EntryChunk))
)


class AmiSymbolFacade:
    def __init__(self, binary):
        self.data = binary
        pass

    def __setitem__(self, key, item):
        self.__dict__[key] = item

    def __getitem__(self, key):
        return self.__dict__[key]

    def __repr__(self):
        return repr(self.__dict__)

    def __len__(self):
        return len(self.__dict__)

    def __delitem__(self, key):
        del self.__dict__[key]

    def clear(self):
        return self.__dict__.clear()

    def copy(self):
         return self.__dict__.copy()

    def has_key(self, k):
        return k in self.__dict__

    def update(self, *args, **kwargs):
        return self.__dict__.update(*args, **kwargs)

    def keys(self):
        return self.__dict__.keys()

    def values(self):
        return self.__dict__.values()

    def items(self):
        return self.__dict__.items()

    def pop(self, *args):
        return self.__dict__.pop(*args)

    def __cmp__(self, dict_):
        return self.__cmp__(self.__dict__, dict_)
```

```
    def __contains__(self, item):
        return item in self.__dict__

    def __iter__(self):
        return iter(self.__dict__)



class AmiHeaderFacade:
    def __init__(self):
        pass


def reverse_bits(byte_data):
    return int("{:08b}".format(byte_data)[::-1], 2)


def read_date(date_tuple):
    values = int.from_bytes(bytes(date_tuple), "little")
    return {
        YEAR: values >> 52,
        MONTH: (values >> 48) & 0x0F,
        DAY: (values >> 43) & 0x1F,
        HOUR: (values >> 38) & 0x1F,
        MINUTE: (values >> 32) & 0x3F,
        SECOND: (values >> 26) & 0x3F,
        MILLI_SEC: (values >> 16) & 0x3FF,
        MICRO_SEC: (values >> 6) & 0x3FF,
        RESERVED: values & 0xE,
        FUT: values & 0x1,
    }


def read_date_data(entrybin):
    stride = 40
    start = 0
    datapackbytes = zip(
        entrybin[start::stride],
        entrybin[start + 1 :: stride],
        entrybin[start + 2 :: stride],
        entrybin[start + 3 :: stride],
        entrybin[start + 4 :: stride],
        entrybin[start + 5 :: stride],
        entrybin[start + 6 :: stride],
        entrybin[start + 7 :: stride],
    )
    result = [el for el in map(read_date, datapackbytes)]
    return result


def create_float(float_tuple):
    return struct.unpack("<f", bytes(float_tuple))[0]


def float_to_bin(data):
    return bytearray(struct.pack("<f", data))


def date_to_bin(day, month, year, hour=0, minute=0, second=0, mic_sec=0, milli_sec=0):
    result = bytearray(8)
    result[7] = year >> 4
    result[6] = (result[6] & 0x0F) + (year << 4) & 0xF0
    result[6] = (result[6] & 0xF0) + month
    result[5] = (day << 3) + result[5] & 0xF8
    return result
    pass


class AmiSymbolDataFacade:
    def __init__(self, binary=None):
        self.binary = binary
        self._empty = False
        self.stride = OVERALL_ENTRY_BYTES
        self.default_header = b"BROKDAt5SPCE\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\:
        self.header = SymbolHeader.parse(self.default_header)
        self.default_header = bytearray(self.default_header)
        if not binary:
            self._empty = True
            self.binary = self.default_header + bytearray(TERMINATOR_DOUBLE_WORD_LENGTH)
            self.binentries = self.binary[NUM_HEADER_BYTES:]
            self.length = 0
            self.set_length_in_header()
            return

        enough_bytes = len(binary) >= (NUM_HEADER_BYTES + TERMINATOR_DOUBLE_WORD_LENGTH)
        if not enough_bytes:
            self._empty = True
            self.length = 0
            self.set_length_in_header()
            self.binary = self.default_header + bytearray(TERMINATOR_DOUBLE_WORD_LENGTH)
            self.binentries = self.binary[NUM_HEADER_BYTES:]
            return
        self.binary = bytearray(self.binary)
        self.header = SymbolHeader.parse(self.binary)
        self.default_header = SymbolHeader.build(self.header)
        self.binentries = bytearray(binary[NUM_HEADER_BYTES:])
        self.length = (
            len(self.binentries) - TERMINATOR_DOUBLE_WORD_LENGTH
        ) // OVERALL_ENTRY_BYTES
        self.set_length_in_header()

    def set_length_in_header(self):
        self.header["Length"] = self.length
        self.default_header = SymbolHeader.build(self.header)

    def _create_blank_header(self):
        pass

    def __len__(self):
        return self.length

    def __getitem__(self, item):
        if self._empty:
            return []
        if type(item) == int:
            return self._get_item_by_index(item)
        if type(item) == slice:
            result = []
        start = self._convert_to_index(item.start)
        stop = self._convert_to_index(item.stop)
        step = item.step
        if item.step == None:
            step = 1
        for i in range(start, stop, step):
            result.append(self._get_item_by_index(i))

        return result

    def _convert_to_index(self, index):
```

```
        if index >= 0:
            return index
        if index < 0:
            return self.length + index

    def _get_item_by_index(self, item):
        index = item
        if item < 0:
            index = self.length + item
        start = index * self.stride
        date_tuple = self.binentries[start : (start + 8)]
        return {
            **read_date(date_tuple),
            CLOSE: create_float(self.binentries[(start + 8) : (start + 12)]),
            OPEN: create_float(self.binentries[(start + 12) : (start + 16)]),
            HIGH: create_float(self.binentries[(start + 16) : (start + 20)]),
            LOW: create_float(self.binentries[(start + 20) : (start + 24)]),
            VOLUME: create_float(self.binentries[(start + 24) : (start + 28)]),
            AUX_1: create_float(self.binentries[(start + 28) : (start + 32)]),
            AUX_2: create_float(self.binentries[(start + 32) : (start + 36)]),
            TERMINATOR: create_float(self.binentries[(start + 36) : (start + 40)]),
        }

    def __iter__(self):
        pass

    def __iadd__(self, other):
        minute, hour, second, micro_second, milli_second = 0, 0, 0, 0, 0
        if MINUTE in other:
            minute = other[MINUTE]
        if HOUR in other:
            hour = other[HOUR]
        if SECOND in other:
            second = other[SECOND]
        if MICRO_SEC in other:
            micro_second = other[MICRO_SEC]
        if MILLI_SEC in other:
            milli_second = other[MILLI_SEC]

        append_bin = date_to_bin(
            other[DAY],
            other[MONTH],
            other[YEAR],
            hour,
            minute,
            second,
            micro_second,
            milli_second,
        )
        append_bin += float_to_bin(other[CLOSE])
        append_bin += float_to_bin(other[OPEN])
        append_bin += float_to_bin(other[HIGH])
        append_bin += float_to_bin(other[LOW])
        if VOLUME in other:
            append_bin += float_to_bin(other[VOLUME])
        else:
            append_bin += float_to_bin(0)
        if AUX_1 in other:
            append_bin += float_to_bin(other[AUX_1])
        else:
            append_bin += float_to_bin(0)
        if AUX_2 in other:
            append_bin += float_to_bin(other[AUX_2])
        else:
            append_bin += float_to_bin(0)
        if TERMINATOR in other:
            append_bin += float_to_bin(other[TERMINATOR])
        else:
            append_bin += float_to_bin(0)
        self.binentries[
            -TERMINATOR_DOUBLE_WORD_LENGTH:-TERMINATOR_DOUBLE_WORD_LENGTH
        ] = append_bin
        self.length = (
            len(self.binentries) - TERMINATOR_DOUBLE_WORD_LENGTH
        ) // OVERALL_ENTRY_BYTES
        self.set_length_in_header()
        self.binary = self.default_header + self.binentries
        return self


class SymbolConstructFast:
    header = "Header" / Bytes(0x4A0)
    entry_chunk = BitsSwapped(EntryChunk)

    @classmethod
    def parse(self, bin):
        binentries = bin[0x4A0:]
        num_bytes = len(binentries)
        numits, offset = divmod(num_bytes, 0x488)  # bytes
        result = {}
        result["Header"] = self.header.parse(bin[0:0x4A0])
        result["Entries"] = []
        start = 0x4A0 - offset
        numits = numits + 1
        result["Entries"].append(self.entry_chunk.parse(bin[0x4A0:]))
        entrybin = bin[start:]
        for i in range(numits):
            result["Entries"].append(
                [
                    self.entry_chunk.parse(entrybin[(i * 40) : (i * 40 + 40)]),
                    self.entry_chunk.parse(entrybin[(2 * i * 40) : (2 * i * 40 + 40)]),
                    self.entry_chunk.parse(entrybin[(3 * i * 40) : (3 * i * 40 + 40)]),
                    self.entry_chunk.parse(entrybin[(4 * i * 40) : (4 * i * 40 + 40)]),
                    self.entry_chunk.parse(entrybin[(5 * i * 40) : (5 * i * 40 + 40)]),
                    self.entry_chunk.parse(entrybin[(6 * i * 40) : (6 * i * 40 + 40)]),
                    self.entry_chunk.parse(entrybin[(7 * i * 40) : (7 * i * 40 + 40)]),
                    self.entry_chunk.parse(entrybin[(8 * i * 40) : (8 * i * 40 + 40)]),
                    self.entry_chunk.parse(entrybin[(9 * i * 40) : (9 * i * 40 + 40)]),
                    self.entry_chunk.parse(
                        entrybin[(10 * i * 40) : (10 * i * 40 + 40)]
                    ),
                    self.entry_chunk.parse(
                        entrybin[(11 * i * 40) : (11 * i * 40 + 40)]
                    ),
                    self.entry_chunk.parse(
                        entrybin[(12 * i * 40) : (12 * i * 40 + 40)]
                    ),
                    self.entry_chunk.parse(
                        entrybin[(13 * i * 40) : (13 * i * 40 + 40)]
                    ),
                    self.entry_chunk.parse(
                        entrybin[(14 * i * 40) : (14 * i * 40 + 40)]
                    ),
                    self.entry_chunk.parse(
                        entrybin[(15 * i * 40) : (15 * i * 40 + 40)]
                    ),
                    self.entry_chunk.parse(
                        entrybin[(16 * i * 40) : (16 * i * 40 + 40)]
                    ),
                    self.entry_chunk.parse(
```

```
                        entrybin[(17 * i * 40) : (17 * i * 40 + 40)]
                    ),
                    self.entry_chunk.parse(
                        entrybin[(18 * i * 40) : (18 * i * 40 + 40)]
                    ),
                    self.entry_chunk.parse(
                        entrybin[(19 * i * 40) : (19 * i * 40 + 40)]
                    ),
                    self.entry_chunk.parse(
                        entrybin[(20 * i * 40) : (20 * i * 40 + 40)]
                    ),
                    self.entry_chunk.parse(
                        entrybin[(21 * i * 40) : (21 * i * 40 + 40)]
                    ),
                    self.entry_chunk.parse(
                        entrybin[(22 * i * 40) : (22 * i * 40 + 40)]
                    ),
                    self.entry_chunk.parse(
                        entrybin[(23 * i * 40) : (23 * i * 40 + 40)]
                    ),
                    self.entry_chunk.parse(
                        entrybin[(24 * i * 40) : (24 * i * 40 + 40)]
                    ),
                    self.entry_chunk.parse(
                        entrybin[(25 * i * 40) : (25 * i * 40 + 40)]
                    ),
                    self.entry_chunk.parse(
                        entrybin[(26 * i * 40) : (26 * i * 40 + 40)]
                    ),
                    self.entry_chunk.parse(
                        entrybin[(27 * i * 40) : (27 * i * 40 + 40)]
                    ),
                    self.entry_chunk.parse(
                        entrybin[(28 * i * 40) : (28 * i * 40 + 40)]
                    ),
                    self.entry_chunk.parse(
                        entrybin[(29 * i * 40) : (29 * i * 40 + 40)]
                    ),
                ]
            )

        return result

```

consts.py

```
DATEPACKED = "DatePacked"
DAY = "Day"
MONTH = "Month"
YEAR = "Year"
VOLUME = "Volume"
CLOSE = "Close"
OPEN = "Open"
HIGH = "High"
LOW = "Low"
TERMINATOR = "TERMINATOR"
AUX_2 = "AUX2"
AUX_1 = "AUX1"
FUT = "Isfut"
RESERVED = "Reserved"
MICRO_SEC = "MicroSec"
MILLI_SEC = "MilliSec"
SECOND = "Second"
HOUR = "Hour"
MINUTE = "Minute"
```

revbitinteger.py

```
from construct import BitsInteger
from construct import (
    IntegerError,
    stream_read,
    swapbytes,
    bits2integer,
    integertypes,
    integer2bits,
    stream_write,
)


class RevBitsInteger(BitsInteger):
    def _parse(self, stream, context, path):
        length = self.length
        if callable(length):
            length = length(context)
        if length < 0:
            raise IntegerError("length must be non-negative")
        data = stream_read(stream, length, "WhateverPath")
        if self.swapped:
            if length & 7:
                raise IntegerError(
                    "little-endianness is only defined for multiples of 8 bits"
                )
            data = swapbytes(data)
        data = data[::-1]
        return bits2integer(data, self.signed)

    def _build(self, obj, stream, context, path):
        if not isinstance(obj, integertypes):
            raise IntegerError("value %r is not an integer" % (obj,))
        if obj < 0 and not self.signed:
            raise IntegerError("value %r is negative, but field is not signed" % (obj,))
        length = self.length
        if callable(length):
            length = length(context)
        if length < 0:
            raise IntegerError("length must be non-negative")
        data = integer2bits(obj, length)
        if self.swapped:
            if length & 7:
                raise IntegerError(
                    "little-endianness is only defined for multiples of 8 bits"
                )
            data = swapbytes(data)
        data = data[::-1]
        stream_write(stream, data, length, path)
        return obj

```

__init__.py

```
from .ami_bitstructs import create_entry_chunk
from .ami_reader import AmiReader
from .ami_database import AmiDataBase
from .ami_dataclasses import SymbolEntry, SymbolData, Master, MasterData, MasterEntry,SymbolConstruct
from .consts import DATEPACKED, DAY, MONTH, YEAR, VOLUME, CLOSE, OPEN, HIGH, LOW

```

7: