Understanding Financial Market Data in Code: A Walkthrough of and Introduction: The Basic Idea

Getting live financial market data is a simple, two-step process. First, your software has to ask a data server for the specific stock, option, or currency you're interested in. Second, once the server knows what you want, it begins sending you a continuous stream of updates for that item. We'll explore two core data structures that model this process: the LiveSubscription class is how you "ask" for data, and the LiveTick class is the "update" you receive. This guide will break down what each class is and what its most important properties mean for a beginner.

1. The Request: How to Ask for Market Data with

The LiveSubscription class is a straightforward data structure used to tell the data server exactly which financial instrument you are interested in. Think of it as a "request form" where you fill in the details needed to pinpoint one specific item out of millions traded on global exchanges. The properties of this class work together to uniquely identify an instrument, whether it's a currency option like "USDINR APR 84.0 PE" or a commodity future like "GOLD APR 72200 CE".

To make this concrete, let's look at how we would request data for the currency option "USDINR APR 84.0 PE" using the LiveSubscription class:

Property Data Type Purpose Example

Source int An identifier for the data source or provider. This numeric code tells the system which data feed to use. (e.g., 1 for a specific provider)

ExchangeCode int A numeric identifier for the specific market where the instrument is traded, such as the National Stock Exchange (NSE) or Multi Commodity Exchange (MCX). (e.g., 10 for CDS)

ScriptCode string The unique, human-readable trading symbol that identifies a specific instrument, such as a particular stock or options contract. "USDINR APR 84.0 PE"

Now that we understand how to request data for an instrument, let's explore the data we get in return.

2. The Data: Understanding the Update

The LiveTick class is the main data container that holds all the live market information for a subscribed instrument.You can think of it as a detailed "snapshot" of the market at a specific moment. Critically, each LiveTick object includes a ScriptCode property, allowing your application to map the incoming data directly back to your original LiveSubscription request. The server sends these LiveTick objects repeatedly to keep your application updated.

While the full LiveTick class has many properties, we will focus on four of the most fundamental ones for understanding market activity: Ask, Bid, LastPrice, and Volume.

A Note on Precision: In financial programming, accuracy is paramount. You'll notice that all price and volume properties in LiveTick use the decimal data type. This is a deliberate choice. Unlike float or double, which can introduce tiny rounding errors, decimal is designed for high-precision arithmetic, ensuring that monetary calculations are always exact.

2.1. Key Market Terms Explained

Ask

The lowest price a seller is willing to accept for the instrument right now.

The Ask price represents the supply side of the market. If you wanted to buy an instrument immediately, this is the best price currently on offer. It's the "sticker price" set by sellers at this very instant.

Bid

The highest price a buyer is willing to pay for the instrument right now.

The Bid price represents the demand side. If you wanted to sell an instrument immediately, this is the best price you could get from a willing buyer. It reflects what the market is currently "bidding" for the asset.

LastPrice

The price at which the most recent trade occurred.

The LastPrice is a historical fact—it's the price at which the last transaction between a buyer and a seller was completed. This value is often what you see most prominently on financial news tickers, as it shows the most recent valuation agreed upon by the market.

Volume

The total number of shares or contracts traded during the current trading day.

The Volume tells you how much trading activity an instrument has seen throughout the day. A high volume indicates that many shares have changed hands, suggesting a high level of interest (either buying or selling) in the instrument.

Developer's Note: In a real-world application, you would monitor the Ask and Bid prices alongside their corresponding quantities (AskQuantity and BidQuantity, which are also in the LiveTick object). This combination gives you insight into market depth—the volume of buy and sell orders available at the current best prices, which is a key indicator of liquidity.

With these key data points in mind, we can see how a complete picture of market activity is formed.

3. Putting It All Together: The Data Flow

The process is a simple and continuous loop. A trading program first creates and sends a LiveSubscription object to specify what it wants to track (e.g., "I want data for the currency option USDINR APR 84.0 PE on the CDS exchange"). In response, the data server begins pushing a stream of LiveTick objects back to the program. Each LiveTick object is a new snapshot containing the latest market data, including the current Ask price from sellers, the Bid price from buyers, the LastPrice of the most recent trade, and the total Volume for the day.

4. Key Takeaways for Beginners

\* LiveSubscription is the Question: It's how your program specifies precisely which financial instrument you want data for.

\* LiveTick is the Answer: It's the data packet that gives you the latest market snapshot for that instrument, updated continuously.

\* Ask, Bid, and LastPrice are Core Indicators: These three values give you a fundamental understanding of the current supply, demand, and trading activity for an instrument.

###################################################

Briefing Document: Analysis of the "Niku - Master Code" System

Executive Summary

The provided source context outlines the architecture and functionality of a financial market data application named "Niku - Master Code." This system is a desktop application developed using C# and WPF, designed to connect to a specialized financial data feed, process real-time and historical market data, and manage a local cache of this information. The application's core is its ability to integrate with the nimblestream.lisuns.com data service, handling a comprehensive range of data points including standard price ticks, OHLC (Open-High-Low-Close) values, and advanced options metrics known as "Greeks" (Delta, Gamma, Vega, etc.). The system is structured to handle various financial instruments across different exchanges, including currency derivatives, commodities, and equities, and features a modular design for potentially integrating multiple data brokers.

1. System Architecture and Overview

The file structure and source code reveal a sophisticated Windows desktop application designed for financial market analysis.

1.1. Project Structure

The application, named "Niku - Master Code," is organized into several distinct logical directories, indicating a well-defined architecture:

\* Brokers: This directory contains the primary logic for data feed integration. The central class is clsGlobalDataLive.cs, which manages the connection to the NimbleStream service. The presence of other files like clsKite.cs, clsMasterTrust.cs, and clsTrueData.cs suggests a modular design intended to support multiple data sources or brokerage APIs.

\* Models: This folder houses the C# class definitions that serve as data structures for the application. Key models include LiveTick.cs, LiveSubscription.cs, Instruments.cs, and clsOHLC.cs, which define how market data is represented within the system.

\* Classes: Contains general-purpose helper and utility classes, such as clsGenfx.cs and clsLiveFeed.cs.

\* Data and NikuData: These directories contain a large number of .achd files. The filenames (AXISBANK15.achd, GOLD AUG FUT.achd, Nifty 50.achd) strongly indicate that these are local cache files for historical market data, categorized by instrument and timeframe (e.g., 1, 3, 5, 15, 30, 60 minutes, daily, weekly).

\* Instruments: Holds text files (Kite.txt, TrueData.txt) that likely serve as master lists of tradable instruments available from different data sources.

\* User Interface (UI): The project includes .xaml files such as MainWindow.xaml and winOptionChain.xaml, confirming it is a WPF (Windows Presentation Foundation) application with a graphical user interface, including a dedicated window for displaying options chain data.

1.2. Technology and Environment

The project files reveal specific details about the development environment:

\* Framework: .NET

\* Language: C#

\* Application Type: Windows Desktop (WPF)

\* Package Management: Utilizes NuGet, with version 6.14.0 specified in the configuration.

\* Local Paths: Configuration files reference a local user path: C:\Users\VGFX2\.nuget\packages\.

2. Core Component: Global Data Feed Integration

The clsGlobalDataLive.cs class is the system's engine for market data acquisition. It is designed to connect, authenticate, and manage data streams from a third-party service.

2.1. Service Provider Details

The connection is hardcoded to a specific service:

\* Root URL: nimblestream.lisuns.com

\* Port: 4525

\* API Key: 575e2397-93d0-4f9a-a6fd-0819cb31febd

2.2. Key Functionalities

The class implements a complete lifecycle for data management through an event-driven model using the nimbleStreamClient.Client object.

\* Connection Management: Methods such as ConnectGlobal, AuthenticateGlobal, and DisconnectGlobal handle the connection state. The code includes logic for automatically attempting to reconnect if the connection is broken.

\* Real-Time Data Subscription: The SuscribeData and UbSubscribe methods manage subscriptions to real-time tick data for specified instruments. The system maintains a list of currently subscribed instruments in globalInstrumentLst.

\* Historical Data Retrieval: The callHistory and getHistorybyGlobal methods request historical OHLC data. They are capable of fetching data across numerous timeframes, including:

\* Intraday: Tick, 1/3/5/10/15/30 Minutes, 1/2/3/4/6/8/12 Hours

\* EOD: Daily, Weekly, Monthly, Quarterly, Half-Yearly, Yearly

\* API Limitation Handling: The client\_LimitationResult event handler is designed to process messages from the server regarding API usage limits, such as AllowedBandwidthPerHour, AllowedCallsPerMonth, and AllowedExchanges.

3. Data Models and Scope

The system utilizes detailed data models to handle a wide array of financial information, with a clear focus on derivatives trading.

3.1. Data Model

The LiveTick class is the primary structure for holding a snapshot of market data for an instrument. Its comprehensive nature supports complex analysis and trading decisions.

Property Type Description

LastTradeTime DateTime? Timestamp of the last executed trade.

Ask decimal The best asking price.

Bid decimal The best bidding price.

Change decimal The price change from the previous close.

PercentageChange decimal The percentage change from the previous close.

Close decimal The previous day's closing price.

Low decimal The lowest price of the current trading session.

High decimal The highest price of the current trading session.

Volume decimal The total traded volume for the session.

LastQuantity uint The quantity of the last executed trade.

LastPrice decimal The price of the last executed trade.

ScriptCode string A unique code for the instrument.

Open decimal The opening price of the current session.

ExchangeCode int The numerical code for the exchange.

BidQuantity int The quantity available at the best bid price.

AskQuantity int The quantity available at the best ask price.

AveragePrice decimal The volume-weighted average price (VWAP).

initalOI int Initial Open Interest for the session.

currentoi int The current Open Interest.

SymbolId int A unique numerical identifier for the symbol.

Timestamp DateTime The timestamp of the tick data packet.

delta float Greek: Rate of change of the option price relative to the underlying asset's price.

gamma float Greek: Rate of change of Delta.

theta float Greek: Rate of change of the option price over time (time decay).

vega float Greek: Rate of change of the option price relative to volatility.

rho float Greek: Rate of change of the option price relative to interest rates.

IV float Implied Volatility of the option.

3.2. Other Key Models

\* LiveSubscription: A simple class with properties Source, ExchangeCode, and ScriptCode used to define an instrument for a data subscription request.

\* Instruments: A class used internally to manage instrument details received from the server, including TradingSymbol, StrikePrice, LotAmount, and InstrumentType.

3.3. Supported Financial Instruments

The data files provide concrete examples of the instruments the application is designed to track:

\* Currency Derivatives: USD/INR options with various strike prices and expiration dates (e.g., USDINR APR 84.0 PE,10380).

\* Commodities: MCX options for precious metals (e.g., GOLD APR 72200 CE,446059, SILVER APR 100000 CE,433762).

\* Equities and Securities: A diverse list of stocks and other securities with associated IDs (e.g., MARUTI AF,11119, ARIHANT RL,15492, NHAI N1,31038).

#############################################################

Technical Specification: Financial Market Analysis System

1.0 Introduction

This document provides a detailed technical specification for the proprietary Financial Market Analysis System. It outlines the architecture, data models, and core logic of a system designed to process historical market data, primarily Open-High-Low-Close (OHLC) values, to generate a series of predictive indicators and qualitative market theories. The system's primary function is to transform raw price data into actionable, state-based insights.

The intended audience for this specification includes software engineers and systems architects who are responsible for maintaining, extending, or integrating with this system. A clear understanding of the components and data flow described herein is essential for any development or operational activities.

This document is structured to provide a layered understanding of the system. It begins by defining the core analytical concepts that form the intellectual foundation of the system. It then describes the high-level system architecture and data flow, followed by a detailed breakdown of the data structures and enumerations. Finally, it provides in-depth specifications for the main processing classes responsible for quantitative calculation and qualitative interpretation.

2.0 Core Analytical Concepts

A firm grasp of the system's proprietary analytical terminology is essential for comprehending the technical details that follow. This section defines the foundational concepts upon which the entire analytical framework is built. Each concept represents a specific layer of abstraction, moving from raw price calculations to interpretive market states.

\* Channels (Rising and Falling) Channels are the foundational grid for all subsequent analysis. They consist of a series of calculated price bands projected from the previous period's price action. Based on the previous period's High-Low range and its Closing price, a set of Rising Channel (RC) and Falling Channel (FC) levels are calculated using a predefined array of percentages (e.g., 11.8%, 23.6%, 38.2%). A small volatility complement (compl), calculated as a fraction of the previous close, is added to these levels for non-Daily timeframes to account for finer price movements; this adjustment is omitted for the Daily timeframe. These channels define the expected boundaries of price movement and are used as a reference for nearly all other indicators.

\* Director Pattern (DP) The Director Pattern is a key proprietary pattern derived from the interplay between the "Junior Good Director" (JGD) and "Junior Weak Director" (JWD) values of the two most recent prior periods. The relationship between these four values determines a specific named pattern—"22", "31", or "21"—and establishes critical price levels for the upcoming period: the Bull Director Pattern (BDP) and the Weak Director Pattern (WDP). These levels act as primary triggers for trend changes.

\* Trending Leg (TL) The Trending Leg represents the system's stateful interpretation of the current market trend. It is defined by the TRENDINGLEG enumeration, which includes states such as Uptrend Continuation (UTC), Trend Reversal Bullish (TRB), and Trend Reversal Sell (TRS). The system transitions from one state to another based on complex conditional logic that evaluates current price action (High and Low) against the calculated Channel and Director Pattern price levels.

\* Theories Theories are qualitative, human-readable interpretations derived from the quantitative analysis. They are named market conditions, such as "BDP Cut-Cut," "Cutting the Flying Kite in RC," or "Catching the falling Knife," that provide context to the market's state. These theories are generated by evaluating where the Director Pattern's BDP and WDP values land within the calculated channel bands.

\* Opening Analysis (ICRC, ICFC, RMSL) This is the system's methodology for classifying the market's behavior within an initial trading window of a given period (e.g., the first 30 minutes of a daily session). By analyzing the High and Low of this opening window against key channel and operational band levels, the system assigns a classification like "ICRC" (Inside Channel Rising Channel), "ICFC" (Inside Channel Falling Channel), or "RMSL" (Range Mid-Section Line) Higher/Lower, providing an immediate snapshot of opening momentum.

These concepts are implemented through the system's architecture, which translates these analytical principles into a concrete data processing pipeline.

3.0 System Architecture and Data Flow

This section provides a high-level overview of the system's software architecture. The design is based on a layered, pipeline architecture that promotes a clear separation of concerns, decoupling quantitative signal generation from subsequent qualitative interpretation. This architectural choice enhances maintainability by isolating the core mathematical engine from the business logic layer that translates its output into human-readable insights.

The architecture consists of two primary components that work in tandem to produce the final analysis.

\* Core Engine (Market.clsNiku) This class serves as the primary computational engine of the system. It is responsible for all foundational quantitative tasks, including data ingestion from JSON files, assembly of historical data, and the calculation of core numerical indicators. Its key outputs are the market state representations: Channels, the Director Pattern, and the Trending Leg.

\* Interpretation Layer (NiKu.clsRecord) This class functions as the analytical interpretation layer. It consumes the numerical output and state information generated by the clsNiku engine and translates it into qualitative, human-readable insights. Its responsibilities include generating "Theories," calculating operational bands (UB/LB), and classifying the market's opening behavior.

The data flows through these components in a sequential pipeline, transforming raw historical data into layered analytical output.

1. Data Ingestion: The process begins with getAllOHLC reading historical OHLC data for a specific financial instrument from JSON files.

2. History Assembly: getHistoryData structures the raw data into a HistoryData struct, providing the three chronologically most recent data points (current, previous, and pre-previous) required for analysis.

3. Quantitative Calculation (clsNiku):

\* getChannels computes the Rising Channel (RC) and Falling Channel (FC) price levels based on HistoryData inputs.

\* getDirectorPattern computes the BDP, WDP, and other associated DP values using the two most recent prior periods from HistoryData.

\* getTrendingLEG, using results from the prior steps, determines the market's current trend state (TRENDINGLEG).

4. Qualitative Interpretation (clsRecord):

\* GetTheories generates qualitative theories by evaluating the position of BDP and WDP relative to channel boundaries.

\* getUBLB computes the operational Upper and Lower Bands for the period based on channel and DP results.

\* OpeningPeriod analyzes finer-grained opening data to classify the market open.

5. Output: The system's final output includes structured data, such as the TLResult struct containing the trend leg and buy/sell signals, and qualitative strings representing the generated Theories and OpenAs classifications.

The following sections will provide a detailed specification of the data structures and classes that implement this data flow.

4.0 Data Structures and Enumerations

The system's data models are critical to its operation. The C# structs, classes, and enumerations defined below represent market data and analytical results as they are processed through the system.

4.1 OHLC Data Models

OHLCdp Class

This class extends a base clsOHLC model to include fields for storing the results of the analytical process directly alongside the price data for a given period.

Property Type Description

Date DateTime The timestamp for the OHLC data period.

Open double The opening price of the period.

High double The highest price of the period.

Low double The lowest price of the period.

Close double The closing price of the period.

Dpleg string Stores the calculated Director Pattern leg (the TRENDINGLEG state).

Dp string Stores the raw Director Pattern string representation.

ValidBS string Stores the "Valid Buy/Sell" signal state.

PCG string Stores the "Post Closing Gap" classification.

4.2 Core Analytical Structs

HistoryData Struct

This struct is fundamental to the system's calculations, providing a convenient three-period lookback window.

Property Type Description

current OHLCdp The OHLC and analytical data for the current period under analysis.

prev OHLCdp The data for the period immediately preceding current.

prev2 OHLCdp The data for the period two steps prior to current.

DirectorPattern Struct

This struct encapsulates all calculated values related to the proprietary Director Pattern.

Property Type Description

JGD double The calculated Junior Good Director price level.

JWD double The calculated Junior Weak Director price level.

BDP double The Bull Director Pattern; a key upside price level.

WDP double The Weak Director Pattern; a key downside price level.

L1 string A formatted string representing the first line of the pattern display.

L2 string A formatted string representing the second line of the pattern display.

PatternName string The name of the determined pattern ("22", "31", or "21").

NewBDP double A projected BDP level, calculated by adding a percentage of the range to the BDP.

NewWDP double A projected WDP level, calculated by subtracting a percentage of the range from the WDP.

channel Struct

This struct holds the collections of calculated values for a single channel type (either Rising or Falling).

Property Type Description

val IList A list of the primary channel price levels.

tvhs IList A list of the "Trading Value High Side" price levels associated with the channel.

tvls IList A list of the "Trading Value Low Side" price levels associated with the channel.

TLResult Struct

This struct serves as the primary return type for the trend analysis, containing the definitive state for a given period.

Property Type Description

TL TRENDINGLEG (enum) The primary TRENDINGLEG state. Stored as an integer.

TL2 TRENDINGLEG (enum) A secondary TRENDINGLEG state, used for intermediate checks in the logic. Stored as an integer.

ValidBs int The valid buy/sell signal state (0: None, 1: Buy, 2: Sell).

TVHSTVLS Struct

This struct serves as a performance optimization by encapsulating the immediate environmental price context for the trend state machine. It pre-calculates and caches all relevant TVHS/TVLS price thresholds for a given period, preventing redundant calculations within the checkTL method's complex logic.

Property Type Description

BDPhL double[] The TVHS/TVLS levels associated with the BDP value.

JWDhL double[] The TVHS/TVLS levels associated with the JWD value.

WDPhL double[] The TVHS/TVLS levels associated with the WDP value.

JGDhL double[] The TVHS/TVLS levels associated with the JGD value.

HighHL double[] The TVHS/TVLS levels associated with the period's HighestHigh price.

LowHL double[] The TVHS/TVLS levels associated with the period's LowestLow price.

4.3 Enumerations

TRENDINGLEG Enum

This enumeration defines the possible states of the market trend as determined by the system's finite state machine. The precise meaning of each state is defined by its entry and transition conditions within the checkTL method.

Member Description

TRB Trend Reversal Bullish: Initial bullish state entered when the current High exceeds the TVHS of the Bull Director Pattern (H > tvhstvls.BDPhL[0]).

UTC Uptrend Continuation: Continuation of a bullish trend, transitioned from TRB when the current High exceeds the TVHS of the period's HighestHigh (H > tvhstvls.HighHL[0]).

cs A corrective state within a bullish trend, entered from TRB or UTC when the Low drops below the TVLS of the JWD if the PatternName is "22".

CSCOUB A breakout state from a cs condition, entered when the High exceeds the TVHS of the BDP.

CSCOUBC A continuation state following a CSCOUB breakout, entered when the High exceeds the TVHS of the period's HighestHigh.

TRS Trend Reversal Sell: Initial bearish state entered when the current Low drops below the TVLS of the Weak Director Pattern (L < tvhstvls.WDPhL[1]).

DTC Downtrend Continuation: Continuation of a bearish trend, transitioned from TRS when the current Low drops below the TVLS of the period's LowestLow (L < tvhstvls.LowHL[1]).

PBB A corrective state within a bearish trend, entered from TRS or DTC when the High exceeds the TVHS of the JGD if the PatternName is "31".

PBBCODB A breakdown state from a PBB condition, entered when the Low drops below the TVLS of the WDP.

PBBCODBC A continuation state following a PBBCODB breakdown, entered when the Low drops below the TVLS of the period's LowestLow.

The clsNiku class, detailed next, is responsible for populating and utilizing these data structures to perform its core calculations.

5.0 Core Engine Specification: Market.clsNiku

The clsNiku class is the central processing engine of the Financial Market Analysis System. It is responsible for all primary quantitative calculations, from retrieving raw historical data to determining the final market trend state.

5.1 Configuration and Properties

The class utilizes several static fields for configuration.

Field Type Purpose

PER double[] An array of percentage values (11.8, 23.6, 38.2, etc.) that form the mathematical basis for all Rising and Falling Channel calculations.

dbPostFix static string[] A legacy field for database table naming conventions.

tblpostfix static string[] A static array of postfixes for constructing table names based on timeframes.

dbPreFix static string A static prefix used in constructing database names.

dailyCheckCP static TimeFrame A configuration property specifying the default timeframe (e.g., M30) for certain internal checks.

5.2 Key Method Specifications

The following methods constitute the core analytical pipeline of the clsNiku engine.

getHistoryData(string Script, DateTime date, TimeFrame p)

\* Purpose: This method is the primary entry point for fetching historical data. Its required intent is to assemble the three chronologically most recent data points ending on the specified date to provide the lookback window (current, prev, prev2) for all downstream calculations.

\* Logic:

\* The method calls getAllOHLC to retrieve the complete OHLC history for the specified script and timeframe.

\* It filters this list for records up to and including the specified date. The implementation's use of Take(3) without an explicit descending sort introduces ambiguity; however, the correct behavior, as seen in related methods like getPrevHistoryData, involves taking the three most recent records.

\* The data is mapped such that the most recent record populates hd.current, the second-most recent populates hd.prev, and the third-most recent populates hd.prev2.

\* Returns: An instance of the HistoryData struct.

getChannels(HistoryData hd, TimeFrame p)

\* Purpose: This method calculates the foundational Rising Channel (RC) and Falling Channel (FC) price levels for the current period.

\* Logic:

\* The calculation is critically based on the High, Low, and Close of the previous period (hd.prev).

\* It iterates through the percentages defined in the PER array.

\* For each percentage, it calls the helper methods getRC and getFC to calculate the primary channel lines.

\* It also computes the associated tvhs (Trading Value High Side) and tvls (Trading Value Low Side) values, which serve as finer-grained threshold levels. A volatility complement is applied to these thresholds for non-Daily timeframes.

\* Returns: An array of two channel structs, where ret[0] contains the Rising Channel and ret[1] contains the Falling Channel.

getDirectorPattern(HistoryData hd)

\* Purpose: This method calculates the proprietary Director Pattern based on a two-period lookback, yielding key price levels and a pattern classification.

\* Logic:

\* It calculates the JGD (Junior Good Director) and JWD (Junior Weak Director) values for both the previous (hd.prev) and pre-previous (hd.prev2) periods.

\* A series of conditional checks evaluate the relationships between these four values (JGD, JWD, JGDprev, JWDprev).

\* Based on these conditions, the method determines the PatternName ("22", "31", or "21") and calculates the final BDP (Bull Director Pattern) and WDP (Weak Director Pattern) price levels.

\* Returns: A populated DirectorPattern struct containing all calculated values.

getTrendingLEG(HistoryData hd, TLResult prevResult, ...) and checkTL(HistoryData hd, ...)

\* Purpose: This method pair implements a classic finite state machine (FSM) to determine the market's trend state (TRENDINGLEG). getTrendingLEG serves as the high-level wrapper that sets up the context, while checkTL contains the core state transition logic.

\* Logic:

\* This is a stateful calculation that requires the previous period's trend result (prevResult) as a critical input to determine the current state.

\* The checkTL method implements the FSM through a series of conditional statements. It checks the current period's High and Low prices against numerous threshold values derived from the Director Pattern and Channels, which are pre-calculated and passed in the tvhstvls struct.

\* These checks determine whether the TRENDINGLEG state should transition to a new state (e.g., from TRB to UTC) or remain the same, as defined in Section 4.3.

\* The method also calculates the ValidBs (Valid Buy/Sell) signal based on the current trend state and whether price has traded beyond certain secondary thresholds.

\* Returns: A TLResult struct containing the new TL state and the ValidBs signal.

getAllOHLC(string script, TimeFrame cp, bool blnNikuData = true, ...)

\* Purpose: This is the foundational data retrieval method for the entire system, responsible for loading raw data from persistent storage.

\* Logic:

\* The method constructs a file path dynamically based on the script, timeframe, and other parameters.

\* It reads the contents of a JSON file from this path.

\* It uses JsonConvert.DeserializeObject to load the raw JSON data into a List<OHLCdp>.

\* A key feature of this method is its ability to merge this primary OHLC data with data from separate Director Pattern value files (dpvals), enriching the base price data with pre-calculated analytical values.

\* Returns: A List<OHLCdp> containing the complete historical data for a given script and timeframe.

The numerical outputs and states generated by clsNiku are then passed to the clsRecord class for higher-level interpretation.

6.0 Interpretation Layer Specification: NiKu.clsRecord

The clsRecord class functions as the system's qualitative analysis layer. Its primary role is to translate the raw numerical outputs and state determinations from the clsNiku engine into descriptive market theories, operational bands, and classifications that are more easily understood by a human analyst.

6.1 Key Method Specifications

GetTheories(double WDP, double BDP, double JGD, double JWD, TimeFrame p, clsNiku.HistoryData hd)

\* Purpose: This method generates a set of named market "theories" by evaluating the positions of the key Director Pattern values (BDP and WDP) relative to the calculated channel levels, providing a qualitative narrative for the current market setup.

\* Logic:

\* The method executes a series of if statements. Each statement checks if the BDP or WDP value falls within a specific price band defined by the channel levels (e.g., BDP < channel[0].val[3] && BDP >= channel[0].val[2]).

\* When a condition is met, a corresponding descriptive string, such as "BDP Cut-Cut" or "Catching the falling Knife", is added to a list of applicable theories.

\* Returns: A formatted string containing all generated theories for the given inputs.

getUBLB(clsNiku.channel[] channel, clsNiku.DirectorPattern dp)

\* Purpose: This method calculates the operative Upper Band (UB) and Lower Band (LB) that are used as key thresholds for the current period's analysis, defining the primary expected boundaries for initial price action.

\* Logic:

\* The method evaluates the DirectorPattern (dp) values (WDP, BDP, PatternName) against the channel structure.

\* By default, the UB and LB are based on the 38.2% channel levels (rc38tvhs, fc38tvls).

\* The bands are widened to the 61.8% levels under specific conditions. For example, the UB is widened to the 61.8% level (channel[0].tvhs[3]) primarily when a "BDP Cut-Cut" condition is met, defined as the BDP falling between the 38.2% and 61.8% Rising Channel levels.

\* Returns: A double array of four values representing UB, UBtvls (the lower side of the upper band), LB, and LBTVHS (the higher side of the lower band).

OpeningPeriod(...) and getOpeningDateAndOpenAS(...)

\* Purpose: These methods work in concert to analyze and classify the market's opening behavior for a given period, thereby categorizing the initial momentum of a trading session.

\* Logic:

\* The methods retrieve fine-grained historical data (e.g., M1 or M15 timeframe) for the opening window of a larger timeframe period (e.g., the first 30 minutes of a Daily bar).

\* It then checks the High and Low of this opening period against the pre-calculated channel values and the operational UB/LB from getUBLB.

\* Based on which thresholds are breached, a classification is derived, such as "ICRC," "ICFC," "RMSL HIGHER," "RMSL LOWER," or "ICRR."

\* If no classification is determined in the initial time window, the method iteratively calls itself to analyze the subsequent time window until a definitive opening pattern is found.

\* Returns: A string array containing the final opening classification (e.g., "ICRC") and the timestamp of the event that determined the classification.

7.0 Conclusion

The Financial Market Analysis System provides a sophisticated, multi-layered framework for interpreting market behavior. The architecture demonstrates a clear separation of concerns, moving logically from raw data ingestion to quantitative indicator calculation within the clsNiku core engine, and finally to qualitative, descriptive analysis in the clsRecord interpretation layer. This specification details the proprietary logic, data models, and processing flow, serving as a foundational technical guide for the ongoing development, maintenance, and integration of the system.