Amplía la Capacidad de tu DARWIN: Fragmenta tus Órdenes de MT5 en Tres Líneas de Código

Si eres proveedor de un DARWIN, la capacidad es el número que decide hasta dónde puedes escalar tu estrategia. Es el techo máximo de capital inversor que tu DARWIN puede absorber antes de que su rendimiento empiece a degradarse bajo el peso de su propia ejecución. Es también uno de los ejes que el equipo de Darwinex Labs utiliza para decidir las asignaciones en INDX entre todos los DARWINs. Una capacidad baja limita tu asignación independientemente de lo buena que sea tu estrategia.

La mecánica es implacable. Tu cuenta de trading subyacente lanza una orden. Esa orden es procesada por el Motor de Riesgo y replicada, con el escalado correcto, en cada cuenta de inversor asignada a tu DARWIN, incluida la de INDX. Lo que en tu lado era una orden de 0.30 lots se convierte, en conjunto, en una orden mucho mayor que puede impactar al mercado.

Esa orden de mercado a tamaño completo puede consumir la liquidez de varios niveles del libro de órdenes. Cuanto más crecen tus activos bajo gestión (AuM), más se dispersan los *fills* (momento exacto en el que una orden se completa), y más ventaja de tu estrategia se transfiere silenciosamente a los proveedores de liquidez del otro lado. Cada pip de slippage se traduce, de forma agregada en todas las réplicas de tu estrategia, en un *capacity score* y un techo de AuM más bajos.

La solución es aparentemente sencilla: no envíes una orden grande; envía varias más pequeñas, espaciadas unos segundos, para dar tiempo al libro de órdenes a reponerse entre los *fills*. De hecho, así funcionan los algoritmos de ejecución institucionales diseñados para reducir el impacto de mercado cuando se operan grandes volúmenes; es la idea básica detrás de la **ejecución *TWAP*** (precio medio ponderado en el tiempo). Aplicado a un DARWIN, eleva directamente el techo de capacidad: la misma estrategia, ejecutando las mismas señales, pero con un huella de deslizamiento materialmente menor por unidad de AuM.

El problema es que implementar esto en MQL5 no es sencillo. Necesitas una *máquina de estados* para rastrear las órdenes *hijas*, gestionar los tiempos, manejar fallos, gestionar la salida y hacer bien el cálculo de los lotes. Es el tipo de código que es tedioso de escribir, fácil de equivocarse e idéntico en cada Asesor Experto que lo necesita.

Por eso hemos construido una pequeña librería *open-source* que se encarga de todo, diseñada específicamente para proveedores de DARWIN. Cambias tres líneas en tu Asesor Experto y la fragmentación ocurre de forma transparente.

⚠ ***Software en fase Alpha. Úsalo bajo tu responsabilidad.***

Este código está en fase alpha y no ha sido probado de forma exhaustiva en distintos brokers, símbolos o condiciones de mercado reales. Es tu responsabilidad revisar el código fuente, ejecutar la librería en una cuenta demo y verificar su comportamiento de extremo a extremo antes de desplegarlo con fondos reales.

Los autores y colaboradores no aceptan ninguna responsabilidad por las pérdidas que pudieran ocasionarse, errores de ejecución, fills parciales, cierres fallidos, deslizamiento, rechazos del broker ni ninguna otra consecuencia financiera u operativa derivada del uso de este software. La librería se proporciona sin garantías de ningún tipo. Consulta las Secciones 7 y 8 de la Licencia Apache 2.0 del repositorio para ver el texto completo del descargo de responsabilidad.

Qué hace la librería

Le indicas el tamaño de lote deseado y fragmenta la ejecución en N órdenes iguales lanzadas de forma secuencial con un retraso configurable. Este es el ciclo de vida completo:

En entrada

Tu Asesor Experto calcula su tamaño de lote habitual, por ejemplo 0.30 lotes. La librería lo fragmenta en N partes (por ejemplo 3), respetando el lote mínimo y el *lot step* del símbolo subyacente. Tu Asesor Experto envía la primera orden con

normalidad. La librería lanza automáticamente las dos restantes, con 10 segundos de separación, cada una con el mismo *stop loss* (SL) que la original.

En salida

La librería monitoriza tu primera posición. Si toca su SL, cada posición hija tiene su propio SL ya enviado al servidor de trading, por lo que se cierran de forma independiente sin que se requiera nada especial. Pero si cierras mediante Take Profit (TP) o manualmente, la librería lo detecta y cierra secuencialmente las posiciones hijas restantes con el mismo retraso. Esto evita también el slippage o deslizamiento en el lado de la salida, algo que a menudo se olvida y que también cuenta para la nota sobre la capacidad (capacity score) de tu DARWIN.

En caso de fallo

Si una orden hija recibe un *requote* (recotización) o falla por cualquier motivo, la librería vuelve a intentar un número configurable de veces antes de omitir ese tramo y continuar. El volumen del tramo fallido queda registrado para que sepas exactamente qué ocurrió.

Instalación

Copia la carpeta **Include/SplitOrder/** en tu directorio de datos de MQL5, dentro de **Include/**. Esto te da tres archivos:

- **SplitOrder.mqh**: la librería principal
- **SplitOrderConfig.mqh**: el *struct* de configuración
- **SplitOrderSQX.mqh**: el adaptador para StrategyQuant X (ignóralo si no usas SQX)

Eso es todo. Sin DLLs, sin dependencias, sin pasos de compilación adicionales más allá de tu build habitual del Asesor Experto.

Integración en tu EA

Veamos un ejemplo real. Imagina que tu EA actualmente abre una compra así:

C/C++

```
void OpenBuy() {
   double lots = 0.30;
   double ask  = SymbolInfoDouble(Symbol(), SYMBOL_ASK);
   double sl   = NormalizeDouble(ask - 200 * _Point, _Digits);

   MqlTradeRequest req = {};
   MqlTradeResult  res = {};

   req.action       = TRADE_ACTION_DEAL;
   req.symbol       = Symbol();
   req.volume       = lots;
   req.type         = ORDER_TYPE_BUY;
   req.price        = ask;
   req.sl           = sl;
   req.deviation    = 5;
   req.magic        = 12345;
   req.type_filling = ORDER_FILLING_IOC;
```

```
   OrderSend(req, res);
}
```

La misma función con el splitting habilitado. Las líneas marcadas con ★ son las únicas adiciones:

```
C/C++
#include <SplitOrder/SplitOrder.mqh>     // ★ Incluye la librería

SplitConfig splitCfg;                    // ★ Declara una vez globalmente
SplitState  splitLong;
SplitState  splitShort;

void OpenBuy() {
   double lots = 0.30;

   // ★ Paso 1: Ajusta el tamaño de lot para pos0
   lots = SplitAdjustLots(Symbol(), lots, splitCfg);

   double ask  = SymbolInfoDouble(Symbol(), SYMBOL_ASK);
```

```
double sl   = NormalizeDouble(ask - 200 * _Point, _Digits);


MqlTradeRequest req = {};
MqlTradeResult  res = {};


req.action       = TRADE_ACTION_DEAL;
req.symbol       = Symbol();
req.volume       = lots;             // Ahora contiene la parte de pos0
req.type         = ORDER_TYPE_BUY;
req.price        = ask;
req.sl           = sl;
req.deviation    = 5;
req.magic        = 12345;
req.type_filling = ORDER_FILLING_IOC;


if(!OrderSend(req, res)) return;


// ★ Paso 2: Registra el fill. La librería se encarga del resto
SplitRegister(splitLong, res.order, splitCfg, 0.30);
// Nota: pasa los lots TOTALES ORIGINALES (0.30), no la cantidad ajustada
```

```
}
```

Luego en tu OnInit(), configura el struct e inicia un timer de 1 segundo:

```
C/C++
int OnInit() {
   SplitConfigInit(splitCfg);
   splitCfg.splitCount   = 3;     // Divide en 3 órdenes
   splitCfg.delaySeconds = 10;    // 10 segundos entre cada una
   splitCfg.magic        = 12345;

   SplitReset(splitLong);
   SplitReset(splitShort);
   EventSetTimer(1);

   return INIT_SUCCEEDED;
}
```

Y en OnTimer(), llama al motor:

```
C/C++
void OnTimer() {
```

```
   SplitManage(splitLong,  splitCfg);
   SplitManage(splitShort, splitCfg);
}
```

Eso es toda la integración. Tu lógica de señales, gestión de riesgo y todo lo demás queda exactamente igual.

¿Usas la clase CTrade? Funciona sin cambios

Muchos EAs modernos usan la clase CTrade de **<Trade/Trade.mqh>** en lugar de construir structs MqlTradeRequest manualmente. Si es tu caso, buenas noticias: la librería no depende de cómo coloques tu primera orden. Solo necesita el ticket resultante.

El mismo patrón de integración usando CTrade:

```C/C++
#include <Trade/Trade.mqh>
#include <SplitOrder/SplitOrder.mqh>

CTrade      trade;
SplitConfig splitCfg;
SplitState  splitLong;
```

```
void OpenBuy() {
   double totalLots = 0.30;
   double pos0Lots  = SplitAdjustLots(Symbol(), totalLots, splitCfg);  // ★
   double ask       = SymbolInfoDouble(Symbol(), SYMBOL_ASK);
   double sl        = NormalizeDouble(ask - 200 * _Point, _Digits);

   if(!trade.Buy(pos0Lots, Symbol(), ask, sl, 0))
      return;

   SplitRegister(splitLong, trade.ResultOrder(), splitCfg, totalLots); // ★
}
```

Las mismas tres líneas de cambio, el mismo comportamiento. Internamente, la librería usa MqlTradeRequest para sus órdenes hijas en lugar de tomar prestada tu instancia de CTrade. Esto mantiene limpios tus valores trade.ResultXxx(), de modo que cualquier código que compruebe trade.ResultRetcode() o trade.ResultPrice() después de tu buy/sell sigue reflejando tu orden, no la actividad interna de la librería.

Filling modes: gestión automática

Un detalle importante: distintos instrumentos admiten distintos modos de filling. Algunos requieren IOC (Immediate-or-Cancel), otros FOK (Fill-or-Kill), algunos

símbolos de exchanges usan RETURN por defecto. Configurarlo mal resulta en órdenes rechazadas con errores de *Invalid fill policy*.

La librería lee la propiedad SYMBOL_FILLING_MODE de cada símbolo y selecciona automáticamente el modo correcto: IOC si está disponible, FOK como alternativa, RETURN como último recurso. Es la misma lógica de detección que usa internamente CTrade::SetTypeFillingBySymbol(), por lo que funciona de forma fiable en todos los símbolos del feed de Darwinex.

No hay nada que configurar. Simplemente funciona.

¿Usas StrategyQuant X? Integración en cuatro pasos

Una gran parte de los proveedores de DARWIN generan sus estrategias con StrategyQuant X (SQX). Cada estrategia generada por SQX enruta todas sus órdenes a través de una única función llamada openPosition(), y en las builds recientes de SQX esa función tiene una firma estable.

Esa consistencia nos permite ofrecer un adaptador específico para SQX. Proporciona una función llamada openPositionSplit() con exactamente la misma firma que openPosition() de SQX, lo que significa que la integración se reduce a un find-and-replace. Cuatro pasos, todos mecánicos:

```C/C++
// 1. Añade el include al inicio de tu archivo .mq5:

#include <SplitOrder/SplitOrderSQX.mqh>



// 2. En OnInit(), añade:

SQXSplitInit(3, 10);

EventSetTimer(1);
```

```
// 3. En OnTimer(), añade:
SQXSplitOnTimer();

// 4. Find-and-replace:
openPosition( → openPositionSplit(
```

El adaptador gestiona toda la complejidad internamente. No declaras variables de estado, no llamas a funciones de registro, no gestionas ningún struct de configuración. La decisión de dividir o no una orden concreta se toma automáticamente según su tipo:

- Las entradas de mercado se fragmentan.
- Las órdenes pendientes (BUY_STOP, SELL_LIMIT, etc.) pasan sin cambios. Dividir una orden pendiente en N órdenes pendientes al mismo precio no reduce el slippage, ya que todas se ejecutan en el mismo instante al tocarse el nivel.
- Las órdenes de salida (isExitLevel = true) pasan sin cambios. Son típicamente cierres EOD, stop-reverses o salidas por señal: situaciones en las que la posición se deshace con urgencia y dividirla dejaría exposición parcial durante la ventana de cierre. No merece la pena el riesgo.
- Las órdenes demasiado pequeñas para dividirse de forma útil pasan sin cambios.

Lo fundamental es que pos0 sigue pasando por el propio openPosition() de SQX, de modo que todas las protecciones nativas de SQX permanecen intactas: comprobaciones de margen, detección de duplicados, el gate

sqHandleTradingOptions (filtro de fin de semana, corte EOD, horario de sesión, máximo de operaciones por día), lógica de reemplazo de órdenes pendientes y el mecanismo de reintento propio de SQX. No sustituimos nada de eso, solo lo envolvemos.

Si usas un LLM como Claude o ChatGPT para ayudarte a modificar tu estrategia, puedes pasarle exactamente esos cuatro pasos y los aplicará correctamente. La transformación es suficientemente mecánica como para que sea difícil equivocarse.

Nota: Las órdenes hijas omiten el gate sqHandleTradingOptions porque son lanzadas directamente por el motor de la librería y no a través de openPosition(). En la práctica, esto solo importa si una orden hija está programada para lanzarse cruzando un límite del gate. Para delays de split típicos de 5–30 segundos, es un caso extremadamente puntual. Si es relevante para tu configuración, evita colocar entradas en el último minuto antes de un corte de SQX.

Cómo funciona el cálculo de lotes

Este es un punto donde los traders suelen tropezar cuando implementan el splitting manualmente, así que vale la pena entender qué hace la librería.

Imagina que quieres 0.30 lotes divididos en 4 partes, y el símbolo usa un lot step de 0.01. La división da 0.075 por tramo, que no es un tamaño de lote válido. La librería aplica un *floor* a cada hijo hasta el step válido más cercano: floor(0.075 / 0.01) * 0.01 = 0.07. Cuatro hijos a 0.07 dan 0.28, por lo que se pierden 0.02 lotes.

La librería resuelve esto asignando el resto a la primera orden (pos0). La distribución real queda así: 0.09 + 0.07 + 0.07 + 0.07 = 0.30. El volumen total siempre se preserva. Esto importa porque un volumen total incorrecto cambiaría el riesgo efectivo en tu cuenta subyacente y, por extensión, representaría mal la señal que se replica en las asignaciones de los inversores.

Si tu volumen total es tan pequeño que el tamaño por hijo caería por debajo del lote mínimo, la librería reduce automáticamente el número de splits hasta que funcione, y registra un mensaje indicando lo ocurrido. Si no puede dividir en

absoluto, vuelve a una orden única. Tu operación sigue ejecutándose, simplemente sin splitting.

Cuando necesitas cerrar todo

Si tu EA tiene un cierre EOD, una reversión de señal o cualquier otra razón para deshacer todas las posiciones de inmediato, llama a SplitCancelAndClose():

```
C/C++
if(SplitIsActive(splitLong))
   SplitCancelAndClose(splitLong, splitCfg);
```

Esto detiene cualquier entrada hija pendiente y cierra secuencialmente todas las posiciones abiertas del split con el mismo delay configurado. El cierre escalonado evita el mismo problema de slippage en el lado de la salida.

Qué ocurre con SL vs. TP

La asimetría en la salida es una decisión de diseño que merece su propia explicación.

Cuando tu primera posición (pos0) toca su Stop Loss (SL), el mercado se mueve en tu contra y la velocidad importa. Cada posición hija tiene su propio SL idéntico configurado en el servidor de trading, por lo que cada una será cerrada de forma independiente por el motor de ejecución del broker: sin delay, sin cierre secuencial y sin intervención de la librería.

Cuando pos0 cierra mediante Take Profit (TP) o cierre manual, la situación es diferente. No hay urgencia, y cerrar todas las posiciones hijas a la vez causaría el

mismo slippage que intentabas evitar. Por eso, la librería inicia una cadena de cierres: cierra una posición hija cada N segundos, igual que las abrió.

Referencia de configuración

Todos los parámetros se encuentran en el struct SplitConfig:

- splitCount (por defecto: 3): Número total de posiciones incluyendo la primera. Usa 2 para una división simple a la mitad, hasta 10 para mayor granularidad. Valores más altos distribuyen el impacto de mercado de forma más uniforme pero también alargan el tiempo total de ejecución.

- delaySeconds (por defecto: 10): Segundos entre cada orden hija, tanto en entrada como en salida. El valor adecuado depende de la liquidez del instrumento. Los pares de FX principales se recuperan en segundos; los cruces exóticos o CFDs con poca liquidez pueden necesitar 15–30 segundos.

- slippage (por defecto: 5.0): Slippage máximo en puntos para las órdenes hijas. Igual que lo que configurarías en un OrderSend normal.

- maxRetries (por defecto: 3): Número de reintentos de una orden hija fallida antes de omitirla.

- retryDelaySeconds (por defecto: 2): Delay entre reintentos, más corto que el delay principal ya que se reintenta el mismo tramo.

- magic: Debe coincidir con el magic number de tu EA para que todas las órdenes hijas estén correctamente atribuidas.

- verbose (por defecto: true): Cuando está activo, imprime logs detallados con el prefijo [SplitOrder][LONG] o [SplitOrder][SHORT] para que puedas rastrear exactamente lo ocurrido en la pestaña Experts.

Pruebas en una cuenta demo

Antes de conectar esto a la estrategia que alimenta tu DARWIN, merece la pena ver la librería en acción con tus propios ojos en una cuenta demo. El repositorio incluye SplitOrder_TestEA.mq5 exactamente para esto.

El EA de prueba abre una operación aleatoria BUY o SELL en EURUSD cada minuto (siempre que no haya ninguna posición abierta) con SL y TP simétricos de 20 pips. Esa simetría es deliberada: obtienes una mezcla aproximada de 50/50 entre salidas por SL y por TP, lo que significa que observarás ambas rutas de cierre sin esperar a una condición de mercado específica.

Adjúntalo a un gráfico de EURUSD en una cuenta demo de Darwinex con los parámetros por defecto (0.05 lots divididos en 3, con 10 segundos de separación) y abre la pestaña Experts. En pocos minutos verás el ciclo de vida completo:

- [TestEA] BUY signal — entry=… pos0=0.03 (total=0.05) — Confirma que pos0 recibió el resto del redondeo (0.03 = 0.01 base + 0.02 resto, los hijos reciben 0.01 cada uno).

- [SplitOrder][LONG] Registered BUY #… — 3 splits, child lots=0.01000… — La librería tomó el control tras el fill de pos0.

- Dos líneas más de Entry X/3 fired espaciadas 10 segundos.

- Cuando la posición se resuelve: bien SL=YES con los hijos cerrándose de forma independiente por sus SLs de broker, bien SL=NO (TP/manual) seguido de la cadena de cierres secuencial.

Es la forma más rápida de que puedas comprobar que la librería hace lo que dice, antes de apuntarla a la cuenta que alimenta tu DARWIN.

Limitaciones actuales

Esta es una versión v1, y hay cosas que intencionadamente no hace:

- Un split activo por dirección: La librería soporta un split activo por dirección. Puedes tener un split long y un split short ejecutándose simultáneamente, pero no dos splits long (piramidar trades). Para la mayoría de estrategias que mantienen una única posición por dirección, esto no es un problema.

- Solo órdenes de mercado: Las órdenes hijas son siempre órdenes de mercado. La librería no soporta dividir una orden pendiente en N órdenes pendientes. La semántica es compleja y el caso de uso es poco frecuente.

- Estado en memoria: El estado sólo se conserva en memoria. Si MetaTrader se reinicia durante un split, las posiciones hijas restantes no se lanzarán. Tu EA tendrá una posición parcial, que puede gestionar mediante su lógica habitual de gestión de posiciones. Es posible que añadamos persistencia basada en fichero en una versión futura.

Por qué es un proyecto comunitario

Hemos construido y publicado esta librería bajo Apache 2.0 porque la capacidad no es una competición de suma cero entre proveedores de DARWIN, sino más bien todo lo contrario.

INDX está creciendo rápidamente, y se alimenta cada vez más de capital institucional. Ese dinero tiene que asignarse en algún lugar, y la capacidad de INDX de ponerlo a trabajar no está limitada por la demanda (hay abundante demanda) sino por la oferta de ventaja invertible. Es decir, DARWINs con suficiente margen de capacidad como para absorber de forma útil una asignación mayor sin que sus costes de ejecución se disparen.

Cuando esa oferta es estrecha, INDX se ve forzado a concentrar la asignación en un puñado de DARWINs con alta capacidad o a reducir su asignación global, y la mayoría de los proveedores nunca ven flujo significativo independientemente de lo interesante que sea su estrategia. Cuando esa oferta es amplia (con muchos

DARWINs habiendo trabajado para reducir su huella de ejecución) INDX puede asignar de forma significativa en más DARWINs, y el flujo total de inversores institucionales hacia estrategias gestionadas por la comunidad crece.

Cada DARWIN que aumenta su capacidad amplía el AUM asignable para todos. El proveedor que implementa *splitting* en su EA no compite contigo. Está expandiendo el techo del flujo que INDX puede poner a trabajar, y una parte de ese incremento acaba llegando a tu balance también.

Por eso esta librería es *open-source* en lugar de una herramienta de pago, y por eso los pull requests son bienvenidos. Si encuentras un error, añades una funcionalidad, la mejora se propaga a todos los que la utilizan. La infraestructura comunitaria mejora cuando la comunidad contribuye.

Obtén el código

La librería es open-source bajo la licencia Apache 2.0. Puedes encontrarla en GitHub en marticastany/darwin-capacity-optimiser, junto con un EA de ejemplo completo que muestra el patrón de integración completo.

Si encuentras bugs o tienes ideas de mejora, los issues y pull requests son bienvenidos. **Solo pedimos que lo pruebes en una cuenta demo primero**. Y si el splitting mueve de forma significativa el capacity score de tu DARWIN, nos encantaría saberlo.

Gracias por leernos,
Martí Castany, Quant — Equipo de Darwinex Labs