```
#!/usr/bin/env python3
"""
ERES PlayNAC KERNEL v2.2
A Biocybernetic Proof-of-Work Runtime for Decentralized Media
Networks

ERES Institute for New Age Cybernetics
Author: Joseph A. Sprute
License: Creative Commons BY-NC 4.0
"""

import numpy as np
import cv2
import hashlib
import time
from typing import Dict, List, Optional, Tuple
from dataclasses import dataclass
from abc import ABC, abstractmethod

#
================================================================================
======
# CORE DATA STRUCTURES
#
================================================================================
======

@dataclass
class MediaTask:
    """Represents a media processing task in the JAS Graph"""
    id: str
    input_frame: np.ndarray
    task_type: str
    nonce: int
    timestamp: float
    ep_value: float = 0.0

@dataclass
class JASLink:
    """JAS Graph edge representing task relationships"""
    source_hash: str
    target_hash: str
    weight: float
    timestamp: float
```

```
    ep_correlation: float

@dataclass
class Block:
    """PlayNAC blockchain block"""
    index: int
    timestamp: float
    media_hash: str
    aura_entropy: float
    ep_value: float
    nonce: int
    previous_hash: str
    hash: str

#
================================================================================
======
# BIOENERGETIC VALIDATION (Bio-PoW Core)
#
================================================================================
======

class AuraScanner:
    """Mock EEG/Biofeedback device interface"""

    def capture(self) -> np.ndarray:
        """Simulate bioenergetic field capture"""
        # In real implementation, this would interface with Muse 2,
NeuroSky, etc.
        return np.random.normal(0.5, 0.1, 256)  # Simulated EEG data

    def is_device_connected(self) -> bool:
        """Check if biofeedback device is available"""
        return True  # Mock implementation

class BioPoW:
    """Bioenergetic Proof-of-Work validator"""

    def __init__(self, gerp_factor: float = 0.618):
        self.scanner = AuraScanner()
        self.gerp_factor = gerp_factor  # Golden ratio for
Vacationomics
        self.entropy_cache = {}

    def generate_ep(self) -> float:
```

```
        """Generate EP (Entropic Potential) value from bioenergetic
data
        EP = Ψ(GERP) × BioEnergetic Entanglement
        """
        if not self.scanner.is_device_connected():
            # Fallback to reduced entropy for non-bio miners
            return np.random.random() * 0.5

        raw_eeg = self.scanner.capture()
        # Calculate spectral entropy
        spectral_entropy = -np.sum(raw_eeg * np.log2(raw_eeg +
1e-10))

        # Apply GERP modulation
        ep_value = spectral_entropy * self.gerp_factor

        # Cache for validation
        timestamp = time.time()
        self.entropy_cache[timestamp] = ep_value

        return ep_value

    def validate_bio_work(self, ep_value: float, network_target:
float, tolerance: float = 0.01) -> bool:
        """Validate bioenergetic proof-of-work"""
        return abs(ep_value - network_target) < tolerance

    def get_aura_entropy(self) -> float:
        """Get current aura entropy measurement"""
        raw_data = self.scanner.capture()
        return -np.sum(raw_data * np.log2(raw_data + 1e-10))

#
================================================================================
======
# MEDIA PROCESSING KERNEL
#
================================================================================
======

class MediaProcessor:
    """Real-time media processing with MD-Complexity validation"""

    def __init__(self, md_complexity_threshold: float = 0.07):
        self.md_complexity_threshold = md_complexity_threshold
```

```
        self.processing_cache = {}

    def calculate_md_complexity(self, frame: np.ndarray) -> float:
        """Calculate MD-Complexity using frame entropy"""
        if len(frame.shape) == 3:
            gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
        else:
            gray = frame

        # Calculate histogram
        hist = cv2.calcHist([gray], [0], None, [256], [0, 256])
        hist_norm = hist.flatten() / hist.sum()

        # Calculate entropy
        entropy = -np.sum(hist_norm * np.log2(hist_norm + 1e-10))
        return entropy / 8.0  # Normalize to [0,1]

    def validate_md_complexity(self, frame: np.ndarray) -> bool:
        """BEE Validation (BioEnergetic Entanglement)"""
        complexity = self.calculate_md_complexity(frame)
        return complexity > self.md_complexity_threshold

    def gerp_transform(self, frame: np.ndarray, ep_value: float) ->
np.ndarray:
        """GERP Media Transformation with EP-adaptive parameters"""
        if not self.validate_md_complexity(frame):
            raise ValueError("MD-Complexity validation failed")

        # EP-adaptive stylization parameters
        sigma_s = 60 + int(ep_value * 100)
        sigma_r = 0.6

        try:
            # Apply stylization
            stylized = cv2.stylization(frame, sigma_s=sigma_s,
sigma_r=sigma_r)
            return stylized
        except Exception as e:
            # Fallback to edge-preserving filter
            return cv2.edgePreservingFilter(frame, flags=1,
sigma_s=sigma_s, sigma_r=sigma_r)

    def process_media_task(self, task: MediaTask) -> np.ndarray:
        """Process media task with validation"""
        frame = task.input_frame
```

```
        # Validate MD-Complexity
        if not self.validate_md_complexity(frame):
            raise ValueError(f"Task {task.id}: MD-Complexity
validation failed")

        # Apply GERP transformation
        result = self.gerp_transform(frame, task.ep_value)

        # Cache result
        self.processing_cache[task.id] = {
            'input_hash':
hashlib.sha256(frame.tobytes()).hexdigest(),
            'output_hash':
hashlib.sha256(result.tobytes()).hexdigest(),
            'ep_value': task.ep_value,
            'timestamp': task.timestamp
        }

        return result

#
================================================================================
======
# JAS GRAPH CONSENSUS
#
================================================================================
======

class JASConsensus:
    """JAS Graph consensus mechanism for task chaining"""

    def __init__(self):
        self.graph = {}  # node_hash -> JASLink
        self.task_history = {}
        self.consensus_threshold = 0.6

    def create_link(self, source_task: MediaTask, target_task:
MediaTask, ep_correlation: float) -> JASLink:
        """Create JAS Graph edge between tasks"""
        source_hash = self._hash_task(source_task)
        target_hash = self._hash_task(target_task)

        link = JASLink(
            source_hash=source_hash,
```

```
            target_hash=target_hash,
            weight=ep_correlation,
            timestamp=time.time(),
            ep_correlation=ep_correlation
        )

        self.graph[f"{source_hash}->{target_hash}"] = link
        return link

    def _hash_task(self, task: MediaTask) -> str:
        """Generate hash for media task"""
        data =
f"{task.id}{task.timestamp}{task.ep_value}{task.nonce}".encode()
        return hashlib.sha256(data).hexdigest()

    def validate_consensus(self, task_hash: str) -> bool:
        """Validate task consensus in JAS Graph"""
        related_links = [link for link in self.graph.values()
                         if link.source_hash == task_hash or
link.target_hash == task_hash]

        if not related_links:
            return True  # Genesis task

        avg_weight = np.mean([link.weight for link in related_links])
        return avg_weight >= self.consensus_threshold

    def get_graph_metrics(self) -> Dict:
        """Get JAS Graph performance metrics"""
        return {
            'total_edges': len(self.graph),
            'avg_weight': np.mean([link.weight for link in
self.graph.values()]) if self.graph else 0,
            'edge_creation_rate': len(self.graph) / max(1,
time.time() - (min([link.timestamp for link in self.graph.values()])
if self.graph else time.time()))
        }

#
================================================================================
======
# PLAYNAC KERNEL (Main Orchestrator)
#
================================================================================
======
```

```
class PlayNACKernel:
    """Main PlayNAC KERNEL orchestrating all components"""

    def __init__(self):
        self.bio_pow = BioPoW()
        self.media_processor = MediaProcessor()
        self.jas_consensus = JASConsensus()
        self.blockchain = []
        self.pending_tasks = []
        self.mining_active = False

    def submit_media_task(self, frame: np.ndarray, task_type: str =
"style_transfer") -> str:
        """Submit new media task for processing"""
        task_id =
hashlib.sha256(f"{time.time()}{task_type}".encode()).hexdigest()[:16]

        task = MediaTask(
            id=task_id,
            input_frame=frame,
            task_type=task_type,
            nonce=0,
            timestamp=time.time(),
            ep_value=0.0
        )

        self.pending_tasks.append(task)
        return task_id

    def mine_block(self, max_iterations: int = 1000) ->
Optional[Block]:
        """Mine a new block using Bio-PoW + Media Processing"""
        if not self.pending_tasks:
            return None

        # Get current task
        task = self.pending_tasks.pop(0)

        # Generate EP value from bioenergetics
        ep_value = self.bio_pow.generate_ep()
        task.ep_value = ep_value

        # Mining loop
        for nonce in range(max_iterations):
```

```
            task.nonce = nonce

            try:
                # Process media task
                processed_frame =
self.media_processor.process_media_task(task)

                # Validate bioenergetic work
                network_target = self._get_network_target()
                if self.bio_pow.validate_bio_work(ep_value,
network_target):
                    # Create block
                    block = self._create_block(task, processed_frame,
ep_value, nonce)
                    self.blockchain.append(block)

                    # Update JAS Graph
                    if len(self.blockchain) > 1:
                        prev_task = self._get_previous_task()
                        if prev_task:
                            self.jas_consensus.create_link(prev_task,
task, ep_value)

                    return block

            except ValueError as e:
                # MD-Complexity validation failed, try next nonce
                continue

        return None  # Mining failed

    def _get_network_target(self) -> float:
        """Calculate current network difficulty target"""
        if not self.blockchain:
            return 0.5  # Genesis target

        # Adaptive difficulty based on recent blocks
        recent_blocks = self.blockchain[-10:]
        avg_ep = np.mean([block.ep_value for block in recent_blocks])
        return avg_ep

    def _create_block(self, task: MediaTask, processed_frame:
np.ndarray, ep_value: float, nonce: int) -> Block:
        """Create new blockchain block"""
```

```
        media_hash =
hashlib.sha256(processed_frame.tobytes()).hexdigest()
        previous_hash = self.blockchain[-1].hash if self.blockchain
else "0" * 64

        block_data =
f"{len(self.blockchain)}{time.time()}{media_hash}{ep_value}{nonce}{pr
evious_hash}"
        block_hash = hashlib.sha256(block_data.encode()).hexdigest()

        return Block(
            index=len(self.blockchain),
            timestamp=time.time(),
            media_hash=media_hash,
            aura_entropy=self.bio_pow.get_aura_entropy(),
            ep_value=ep_value,
            nonce=nonce,
            previous_hash=previous_hash,
            hash=block_hash
        )

    def _get_previous_task(self) -> Optional[MediaTask]:
        """Get the previous task for JAS Graph linking"""
        # In a real implementation, this would retrieve from task
history
        return None

    def get_status(self) -> Dict:
        """Get current kernel status"""
        return {
            'blockchain_height': len(self.blockchain),
            'pending_tasks': len(self.pending_tasks),
            'bio_device_connected':
self.bio_pow.scanner.is_device_connected(),
            'jas_graph_metrics':
self.jas_consensus.get_graph_metrics(),
            'last_ep_value': self.blockchain[-1].ep_value if
self.blockchain else 0,
            'mining_active': self.mining_active
        }

#
==========================================================================
======
# EXAMPLE USAGE & TESTING
```

```
#
================================================================================
======

def demo_playnac_kernel():
    """Demonstration of PlayNAC KERNEL functionality"""
    print("🌀 ERES PlayNAC KERNEL v2.2 Demo")
    print("=" * 50)

    # Initialize kernel
    kernel = PlayNACKernel()

    # Create sample video frame
    sample_frame = np.random.randint(0, 255, (480, 640, 3),
dtype=np.uint8)

    # Submit media task
    task_id = kernel.submit_media_task(sample_frame,
"style_transfer")
    print(f"📥 Submitted media task: {task_id}")

    # Mine block
    print("⛏️  Mining block...")
    block = kernel.mine_block()

    if block:
        print(f"✅ Block mined successfully!")
        print(f"   - Block Index: {block.index}")
        print(f"   - EP Value: {block.ep_value:.4f}")
        print(f"   - Aura Entropy: {block.aura_entropy:.4f}")
        print(f"   - Nonce: {block.nonce}")
        print(f"   - Hash: {block.hash[:16]}...")
    else:
        print("❌ Mining failed")

    # Display status
    status = kernel.get_status()
    print("\n📊 Kernel Status:")
    for key, value in status.items():
        print(f"   - {key}: {value}")

if __name__ == "__main__":
    demo_playnac_kernel()
```

Made with
Artifacts are user-generated and may contain unverified or potentially unsafe content.
Report
Customize Artifact